WL-TR-96-1039

VLSI TESTABILITY SYNTHESIS TOOL

CHIEN-IN HENRY CHEN
JOEL YUEN
RAY YANG

WRIGHT STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL ENGINEERING
DAYTON OH 45435

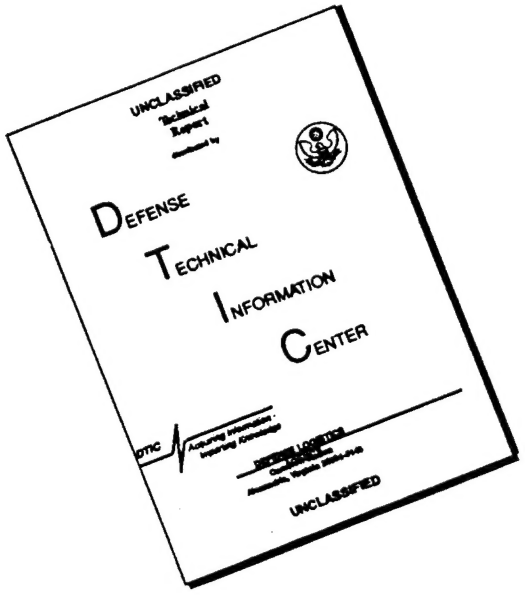
FEBRUARY 1996

FINAL REPORT FOR 04/13/93 -- 02/26/96

Approved for public release; distribution unlimited

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7623

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Mark T. Michael

MARK T. MICHAEL, Project Engineer
Avionics Architecture Technology Branch

JOHN C. OSTGAARD, Chief
Avionics Architecture Technology Branch

ROBERT NIELSEN, Major, USAF
Actg. Chief, System Concepts
& Simulation Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AASA, WPAFB, OH 45433-7334 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	FEB 1996	FINAL 04/13/93--02/26/96

4. TITLE AND SUBTITLE

VLSI TESTABILITY SYNTHESIS TOOL

5. FUNDING NUMBERS

C: F33615-93-C-1226
PE: 62204
PR: 2003
TA: 04
WU: 49

6. AUTHOR(S)

CHIEN-IN HENRY CHEN
JOEL YUEN
RAY YANG

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

WRIGHT STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL ENGINEERING
DAYTON OH 45435

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-77623
POC: Mark Michael, WL/AAST-1, 513-255-7658, ext 4160

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

WL-TR-96-1039

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The VTST project developed a set of Built-In-Self-Test (BIST) methodologies and implemented them in the VTST Computer Aided Design (CAD) toolset. These test methodologies include a pseudo-exhaustive parallel BIST technique that utilizes an efficient test signal reduction method for combinational circuits based on Dr. Chen's research. This technique reduces the size of the test pattern generator (TPG) and the number of test patterns required for a given Circuit Under Test (CUT). Dr. Chen's method was extended to include methods for testing storage elements, i.e., sequential circuits. The VTST toolset includes a non-scan circular BIST method, full and partial-scan methodologies, and circular BIST combined with pseudo-partial scan. Programs are included for fault simulation, partitioning circuits into subcircuits to improve fault coverage, removing redundant faults, synthesizing BIST circuits, and automatically inserting the BIST circuits into the original circuit. The VTST toolset interfaces with LSI Logic's CMDE CAD toolset, generating BIST'ed circuits in LSI's NDL format. A VHDL parser is included that allows VHDL designs to be input to VTST; VHDL output can also be generated. The tools are hosted on a Sun workstation and permit concurrent engineering use on multiple machines. Several test circuits were processed to verify correct operation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	254
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Contents

1 Introduction 1
- 1.1 Background of Testability Synthesis . . . 2
- 1.2 Motivation . . . 4
  - 1.2.1 The Complexity Issue . . . 4
  - 1.2.2 The Quality Issue . . . 5
  - 1.2.3 The Test Generation and Test Application Issues . . . 5
  - 1.2.4 The Economics Issue . . . 6
- 1.3 An Overview of VTST . . . 6
- 1.4 Scope . . . 7

2 Architecture for Automated BIST Design 9
- 2.1 BIST Architecture . . . 11
- 2.2 Stimuli Generation Module . . . 11
- 2.3 Response Analysis and Compression (RA&C) Module . . . 13
- 2.4 System Control Signals and BIST I/O pins . . . 14
- 2.5 Fault Status Generator . . . 17
- 2.6 System Multiplexers . . . 17

3 VTST Front-End Parser 19
- 3.1 VTST System Files . . . 20
  - 3.1.1 Design Files - *.file* . . . 20
  - 3.1.2 Library Technology Files - *.inlib, .outlib* . . . 21
  - 3.1.3 Unknown/Ignored Components . . . 23
- 3.2 Equivalent Logic Cell library . . . 23
- 3.3 Testability Description Netlist (TDN) . . . 25
  - 3.3.1 TDN File Structure . . . 25
  - 3.3.2 Circuit Declaration Section . . . 25
  - 3.3.3 Input/Output Declaration Section . . . 27
  - 3.3.4 Interconnection Section . . . 27
  - 3.3.5 Optional Direct Connection Section . . . 28
  - 3.3.6 Reserved Words . . . 28
- 3.4 NDL Parser . . . 28
  - 3.4.1 A Brief Overview of NDL . . . 30
  - 3.4.2 Data Structure Design . . . 30
  - 3.4.3 System Flow of the NDL Parser . . . 31
- 3.5 VHDL Parser . . . 31

3.5.1 Data Structure for VHDL Parser 32
3.5.2 System Flow of the VHDL Parser 32

4 BIST Design and Testability Analysis Tools: (BISTDaTA) 35
4.1 File Hierarchy of BISTDaTA tools 36
4.2 System Flow 37
4.3 Circuit Preprocessing Stage 37
4.4 BIST/Testability Analysis Stage 37
4.4.1 Testability Analysis for Combinational Circuits 38
4.4.2 Testability Analysis for Sequential Circuits 39
4.4.3 Fault Coverage Estimation 40
4.5 BIST Decision Making Stage 40
4.6 BIST Insertion Stage 41
4.7 Summary 41

5 Circuit Analysis and Preprocessing Tool: Sensor-Scissor 44
5.1 A Tool for Circuit Analysis - Sensor 44
5.2 A Tool for Circuit Preprocess - Scissor 44
5.2.1 Testing Tristate Buffers 46
5.2.2 Testing Tristate Bus 46
5.2.3 Bidirectional Input/Output Ports 48
5.2.4 Gated Clock, Gated Set, Gated Reset Flipflops and Latches 51

6 Pseudo-Scan Processing Tool: Pseudo-Scan Scissor 53
6.1 Pseudo-Combinational Circuit and Pseudo-Scan Design 54
6.2 Pseudo-Scan Algorithm 55
6.3 Pseudo-Scan without Flip-flops and Latches 56
6.4 Pseudo-Scan without Flip-Flops and No Latches in the Feedback Path . . 57
6.5 Pseudo-Scan without Flip-flops and latches in the Feedback Path 58

7 Automated Synthesis Tool for Pseudo-Exhaustive Test Generator: BIST-SYN 59
7.1 Introduction 59
7.2 Phase One: Cluster Partitioning 64
7.3 Phase Two: Formation of Linear Sum of Two Test Signals 72
7.4 Phase Three: Formation of Linear Sum of Multiple Test Signals 75
7.5 Experimental Results 85
7.5.1 Benchmark Circuits 86
7.6 Conclusions 87

8 Circuit Partitioning Tool: AUTONOMOUS 91
8.1 Introduction 91
8.2 Circuit Partitioning 93
8.3 Computing Lower Bound on the Number of Interconnections 99

9 BIST Design Tools for Sequential Circuits: SEQBIST **102**
9.1 Full Scan Design . . . 103
9.2 Partial Scan Design . . . 104
9.3 Circular BIST . . . 105
9.3.1 Introduction . . . 105
9.3.2 CBIST . . . 107
9.4 CBIST with Pseudo-Partial Scan (CBIST w. PPSCAN) . . . 110
9.4.1 Conclusions . . . 111

10 Random Memory Testing **113**
10.1 Introduction . . . 113
10.2 Random Testing for Stuck-at Faults . . . 114
10.2.1 Analysis of the results . . . 119
10.3 Testing for coupling fault . . . 125
10.3.1 Random Test Methodology . . . 125
10.4 LSI BIST Design . . . 130
10.5 Modified Marching 1/0 Test . . . 131
10.6 Transparent BIST RAM . . . 132
10.6.1 Comparison of BIST Techniques . . . 133

11 NDL BIST Automation Tool: BUILDER **138**
11.1 Introduction . . . 138
11.2 BUILDER . . . 139
11.2.1 NDL Parser . . . 139
11.2.2 BIST Results (Actions) Read and Preprocess . . . 140
11.2.3 Circuit Modification . . . 140
11.2.4 BIST Component Creation . . . 141
11.2.5 Internal Connection . . . 142
11.2.6 BISTed Circuit Generation . . . 142
11.3 Usage . . . 142
11.3.1 List of NDL Netlist Files . . . 144
11.3.2 List of BIST Action Files . . . 144
11.3.3 Input Technology Library . . . 144
11.3.4 Unknown Components list . . . 144

12 VTST Test Scheme for Automated BIST Design **145**
12.1 Non-Scan Test Scheme . . . 146
12.1.1 An Example of Non-Scan Test Scheme . . . 149
12.2 Scan Test Scheme . . . 150

13 VTST Distribution System **153**
13.1 Overview . . . 153
13.2 *VMGR* : VTST Tool Manager . . . 154
13.2.1 Manage VTST Tool Server. . . . 154
13.2.2 Manage VTST Tool Client. . . . 155
13.2.3 Register and Deregister VTST Synthesis Tool. . . . 156

13.2.4 Display VTST Tool's Run Time Message. 156
13.3 *VTSVC* : VTST Tool Server. 157
13.4 *VCLNT* : VTST Tool Client 158
13.5 VTST Synthesis Tools 159
13.6 VTST on Local Mode 159
13.7 Running VTST on Multiple Workstations 159
13.8 Summary 160

14 VTST Graphics User Interface **161**
14.1 Overview 161
14.2 Using VMGR 162
14.2.1 How to Start VTSVC 162
14.2.2 How to Start VCLNT 164
14.2.3 How to View a Tool's Run Time Message 164
14.2.4 How to Remove a Tool's Run Time Message 164
14.2.5 How to Quit 164
14.3 Using VCLNT : Environment Set Up 164
14.3.1 Set Current Working Directory 164
14.3.2 Set Tech Library 165
14.3.3 Set Net List File 167
14.3.4 Set Action Sequence 168
14.3.5 Set Unknown Component List 169
14.4 Using VCLNT : Tool Invocation 170
14.4.1 Parser 171
14.4.2 BUILDER 173
14.4.3 Fault Simulator 174
14.4.4 ATPG 177
14.4.5 TPG 178
14.4.6 SEQBIST 179
14.4.7 Autonomous 180
14.4.8 BISTSYN 181
14.4.9 Scissor 183
14.4.10 Action Generator 184
14.4.11 Xor Tree Generator 184
14.4.12 CMDE Simulator Control files Generator 185

15 Test Case: System Coprocessor CP0 of the LR33000 CPU Core **187**
15.1 An Overview of CP0 of the LR33000 CPU Core 187
15.2 VTST Parser on CP0 187
15.3 Circuit Analysis/Preprocessing Tool on CP0 189
15.4 Pseudo-Scan Scissor on CP0 190
15.4.1 Pseudo-Scan with no Flip-Flops and Latches 190
15.4.2 Pseudo-Scan with no Flip-Flops and without Latches in Feedback path 191
15.4.3 Pseudo-Scan with no Flip-Flops and no Latches in Feedback Path . 192
15.5 BISTSYN on CP0 After Pseudo-Scan Scissor on CP0 192
15.5.1 Pseudo-Scan with no Flip-Flops and Latches 192

15.5.2 Pseudo-Scan with no Flip-Flops and without Latches in Feedback Path 193
15.5.3 Pseudo-Scan with no Flip-Flops and no Latches in Feedback Path . 194
15.6 Experimental Results 195

16 TEST CASE: ENHANCED MEMORY CHIP (EMC) 202
16.1 Introduction 202
16.2 EMC top level description 203
16.3 Multiplier Tree Description and Module Testing 205
16.3.1 Multiplier Tree Description 205
16.3.2 Multiplier Tree Module Testing 207
16.4 ALU Block Description and Module Testing 212
16.4.1 ALU Block Description 212
16.4.2 ALU Block Module Testing 216
16.5 Memory Plane, Controller Description and Module Testing 218
16.5.1 Memory Out Controller and Word Decoder Description 218
16.5.2 Testing of Memory Out Controller 218
16.5.3 Memory Plane description 220
16.5.4 Testing of Memory Plane 221
16.6 Testing of Multiplier, ALU Block and Memory Plane at the chip level . . . 221
16.6.1 Testing of Multiplier 222
16.6.2 Testing of ALU Block 223
16.6.3 Testing of Memory Plane 224
16.7 Conclusion 225
16.8 Appendix: VTST Experimental Results 225

Bibliography 234

List of Figures

1.1 Overview of VTST system 6

2.1 BIST architecture 12
2.2 Self terminating test pattern generator 12
2.3 Self terminating test pattern generator (ST-TPG) 13
2.4 Self terminating BILBO cell 14
2.5 Response Analysis and Compression (RA&C) module 14
2.6 Self terminating MISR (ST-MISR) 15
2.7 Self terminating MISR Cell 15
2.8 Fault Status Generator (FSG) 18

3.1 VTST interface with commercial CAD/CAE tools 19
3.2 VTST parsing stage 20
3.3 Example of library technology mapping file 22
3.4 Equivalent logic for OR-AND gate 24
3.5 Testability Description Netlist (TDN) 29
3.6 NDL skeleton files 30
3.7 Data structure for the NDL Parser 31
3.8 Flow chart for NDL parser 32
3.9 Data structure for VHDL parser 33
3.10 Flow chart of the VHDL parser 33

4.1 File hierarchy of BISTDaTA tools 36
4.2 BISTDaTA system flow chart 36
4.3 Circuit preprocess stage 37
4.4 BIST analysis stage 38
4.5 BIST decision making stage 40
4.6 BIST insertion stage 41
4.7 BISTDaTA flow chart 1 42
4.8 BISTDaTA flow chart 2 43

5.1 A window report generated by Sensor-Scissor program 45
5.2 Testable design for a tristate buffer 47
5.3 Tristate bus with inputs and enables from the same source 47
5.4 Tristate bus with inputs from the same source but enables from different source 48

5.5 Tristate bus with inputs from different source but enables from the same source . . . 49
5.6 Tristate bus with inputs and enables from different source . . . 49
5.7 Bidirectional Input/Output Port . . . 50
5.8 Bidirectional Multiplexers . . . 50
5.9 Gated clock D-type flip-flop . . . 51
5.10 Gated clock D-type flip-flop . . . 51
5.11 Gated clock D-type flip-flop . . . 52
5.12 Gated enable D-type latch . . . 52
5.13 Gated set D-type latch . . . 52
5.14 Gated clear D-type latch . . . 52

6.1 Sequential circuit . . . 54
6.2 Pseudo-combinational circuit for test purpose . . . 54
6.3 Pseudo-scan testing . . . 55
6.4 Pseudo-scan without flip-flops and latches . . . 56
6.5 Pseudo-scan without flip-flops and latches in the feedback path . . . 57
6.6 Pseudo-scan with flip-flops and latches in the feedback path . . . 58

7.1 Testing scheme by BISTSYN . . . 60
7.2 Network partition scheme . . . 65
7.3 Ten ISCAS benchmark combinational circuits . . . 66
7.4 Circuit and NA graph of Example 1 . . . 69
7.5 Selection of first candidate pair . . . 70
7.6 Selection of second candidate pair . . . 70
7.7 Selection of third candidate pair . . . 71
7.8 First test scheme of Example 1 . . . 71
7.9 Results of 4 Benchmark circuits by phase one algorithm . . . 71
7.10 NA_1 graph of example 1 . . . 75
7.11 Second test scheme of example 1 . . . 75
7.12 Circuit example 2 . . . 77
7.13 NA graph of example 2 . . . 77
7.14 NA graphs for Example 2 . . . 78
7.15 Formation of linear sum in output dependency sets of example 2 . . . 80
7.16 Test Generator of Example 2 . . . 81
7.17 Test patterns and simulation results of BISTSYN and previous techniques 82
7.18 Flow chart of Three Phase Cluster Partitioning Algorithm . . . 83
7.19 BISTSYN System . . . 84
7.20 Fault simulations of C2670 . . . 85
7.21 Test generation results of benchmarks . . . 86
7.22 Fault simulation results of benchmarks . . . 87
7.23 ATPG and fault simulation results after partitioning . . . 88

8.1 Example of an (8 2 8) circuit . . . 94
8.2 Example circuit . . . 95
8.3 Example 1: graph partitioning . . . 97

8.4 Example 2: graph partitioning 98

9.1 CSTP register cell 106
9.2 CSTP cyclic shift register path 107
9.3 Single tree stage 108
9.4 Dual tree stage with CSTP 108
9.5 CBIST register cell 109
9.6 CBIST w. PPSCAN 111

10.1 An embedded RAM 115
10.2 Markov model for finding stuck-at-0 faults 115
10.3 Markov model for detecting stuck-at-1 faults 118
10.4 Random testing of RAM 120
10.5 Test length for stuck-at-zero vs fault coverage 121
10.6 Test length for stuck-at-one vs fault coverage 122
10.7 Cells initialized to zero 123
10.8 Cells initialized to one 124
10.9 Markov chain model for coupling Faults 126
10.10 Random pattern test length for Coupling Faults 128
10.11 Random pattern test length for coupling Faults 129
10.12 Comparison of LSI-BIST test length with T-BIST and 13N 135
10.13 Coupling fault versus LSI BIST 136
10.14 Comparison of RAM testing techniques 137

11.1 VTST and NDL netlist BUILDER 138
11.2 NDL Netlist BUILDER 139
11.3 User interface for BUILDER 143

12.1 VTST simulation interface 145
12.2 System flow for non-scan test scheme 147
12.3 Timing diagram for non-scan test scheme 149
12.4 Example of timing diagram for non-scan test scheme 150
12.5 System flow for scan test scheme 151
12.6 Timing diagram for scan test scheme 152

13.1 VTST and distribution system 154
13.2 Client/Server relation between VMGR and VTSVC 155
13.3 Client/Server relation between VMGR and VCLNT 156
13.4 Client/Server relation between VMGR and VTST tool 157
13.5 Client/Server relation between VCLNT and VTSVC 158

14.1 VTST tool manager (VMGR) 162
14.2 Start local tool server 162
14.3 Start distributed tool server 163
14.4 Add new host 163
14.5 VCLNT system file pull down menu 165
14.6 VCLNT: set working directory 165

14.7 Directory navigator window 166
14.8 VCLNT: set technology library 166
14.9 Technology library list 167
14.10VCLNT: set netlist 167
14.11Multiple file selection dialog window 168
14.12VCLNT: set BUILDER action sequence 169
14.13VCLNT: set unknown component 169
14.14VCLNT: add new unknown component 170
14.15VCLNT tool menu 170
14.16Single file selection dialog window 171
14.17VTST Parser 172
14.18VTST BUILDER 173
14.19VTST fault simulator 175
14.20VTST fault simulator 177
14.21VTST automatic test pattern generator 178
14.22VTST test pattern generator 179
14.23VTST sequential BIST 180
14.24VTST Autonomous 181
14.25VTST BISTSYN 182
14.26VTST Scissor 183
14.27VTST action generator 184
14.28VTST XOR tree generator 185
14.29VTST CMDE simulation files generator 186

15.1 Block Diagram of the system control of coprocessor (CP0) of the LR33000 CPU 188

16.1 A typical Computer Graphics System Architecture 202
16.2 Basic Architecture of Enhanced Memory Chip 204
16.3 Single Multiplier Stage 205
16.4 Depth 2 Multiplier Stage 206
16.5 Depth 32 Multiplier Stage 208
16.6 Depth 32 Multiplier Stage - continue 209
16.7 Layout of the whole Multiplier Tree 210
16.8 Layout of the whole ALU Block 213
16.9 Layout of the small ALU block 214
16.10Truth Table For AIn, BIn and CIn 215
16.11Block Diagram of Memory Out Controller 219

List of Tables

9.1 Fault coverage of linear tree stage 109
9.2 LSI Logic LCB007 Cell Units for common gates 109
9.3 Circular BIST simulation results for ISCAS-89 benchmarks (20K patterns) 110

10.1 9N Linear Test Algorithm 131
10.2 13N Linear Test Algorithm 132
10.3 Modified Marinescu's Algorithm for Transparent BIST 132
10.4 Signature Prediction Algorithm 133

Chapter 1

Introduction

As digital technology has progressed over the last two decades, dramatic improvements in the architectural design, power consumption, new device development, and other characteristics have enabled the construction of larger and more complex systems into a single chip. The complexity of VLSI chips has increased to a point that designers can no longer carry out their tasks without aid from various design tools. Furthermore, traditional testing techniques are costly and ineffective because of the difficult accessibility of internal circuit nodes (i.e. controllability and observability). The only way to test circuits cost effectively is not simply to use computer-aided design tools, but also to make use of testable structure in the design process - design for testability. Crosscheck Inc. is currently marketing a computer-aided design (CAD) based circuit design system which has a high degree of success in fault detection for it includes automatic inclusion of built-in self-test (BIST) circuitry. However drawbacks to the system relative to the amount of silicon overhead (e.g. 25% for 99% fault detection) have caused many designers to rethink their test requirements. Most research in design synthesis has not included testability consideration. Although many researchers stress that testability should be considered during the early stages of design, most testability research has been done after a structural design solution is defined with no feedback to the original synthesis process for finding more testable designs. Therefore, it becomes very important to develop a supporting set of design synthesis tools which includes testability as an integral part of design as well as efficiently handles the rapidly increasing complexity of the creative design process.

In recent years, there has been considerable interest in automated synthesis at a higher level of the design hierarchy. The reasons are as follows:

- *To shorten the design time.* The economics of producing a digital system weighs heavily in favor of a short design cycle. The cost of the design time itself is a major portion of the final system cost, and the profitability of the system is highly sensitive to how quickly it can be made available to the market.

- *To reduce the design errors.* If the automated design system is itself verified to be correct, then the final design it produces will correspond to its initial specification. This will mean fewer errors and less debugging time for the produced design.

- *To explore the design space.* A good automated design system can produce several

designs for the same specification in a reasonable amount of design time. This allows the designers to choose the best design to meet their constraints.

Traditionally, synthesis techniques have been classified in three levels. The high level is behavioral level synthesis which generates logic representations. The middle level is logic level synthesis which optimizes gate level representations. The low level is physical level synthesis which deals with the geometric view of the chip. The low level synthesis has matured to a point that the designers commonly use tools based on such techniques to analyze and optimize digital designs. In general, logic level synthesis provides two capabilities: technology independent circuit optimization and technology dependent library cells. Most available systems have been conceived for combinational logic design and then extended to support synchronous circuit design. At present, a common way of entering a digital design is to use programs that capture the logic schematic. The designer's major task is to transform the hardware specification into a logic schematic, which is generally validated by simulation. Tools for physical level synthesis are also very common. Several physical level design systems are available in marketing or distributed from universities to support the geometric design of chips. Traditionally the prime objective of layout algorithms is to minimize the total layout area by carefully placing the given modules and making all required interconnects. These placement schemes try to shorten the net lengths among modules by placing the modules with more net connections closer together. However, the circuit performance is decided by a clock rate inserted to the circuit. The maximum clock rate is determined (or bounded) by the longest path delay present in the circuit. Therefore, the performance of a design tends to be path-oriented in nature and the minimization of total net length may not lead to the improvement of the performance. Research in this area is focusing more on the problems of timing related emerging design methodology, for example, timing driven layout.

The high-level synthesis task is to map an input specification for a hardware design into a structure that implements the behavior while satisfying the goals and constraints. The input specification normally contains behavior information. By behavior we mean the way the system or its components interact with their environment, i.e., the mapping from the inputs to outputs. Structure refers to the interconnected components that make up the system. Usually, there are a number of different structures that can be used to realize a given behavior. One of the tasks of high-level synthesis is to find the structure that best meets the constraints such as speed, cost, chip area, pin count, power consumption, testability, test time and design time.

1.1 Background of Testability Synthesis

With the growth in the complexity of VLSI circuits, the cost of testing a chip, too, has become an important part of the overall design and manufacturing cost. The only way to reduce the testing cost is to make use of the testable structure during the design process - design for testability. Testability can be defined as the ease of testing, or the ability to test circuits easily or cost effectively. Testability should be adopted as one of the circuit design parameters by the fact that the cost of hardware is decreasing steadily relative to the cost

of testing.

Approaches to testing can be divided into two categories: External testing and built-in self-test. In external testing, the test patterns are supplied to the circuit under test (CUT) by an external tester that is capable of storing test patterns and the corresponding correct response. However, such external testers are quite expensive. In addition, there are some problems in using external testers:

- *The turnaround time to obtain the test patterns and the computation cost are becoming too long and too high, respectively.*
- *The tester can not efficiently handle a very large number of test patterns.*
- *Testing is too slow because of the reliance on slow test sets to supply the test patterns and analyze the results.*

These problems make the testing cost grow rapidly every year as technological capability increases without a corresponding increase in circuit accessibility. To keep chip testing costs within reason, designers have turned to the use of self-testing techniques. Indeed, Built-In Self-Test (BIST) is proving to be an essential design strategy in many situations. BIST includes on-chip circuitry to provide test patterns and to analyze the output response. It can perform the testing internal to the chip, so that the need for complex external testing equipment is greatly reduced. Using BIST, a significant reduction in test volume can be achieved, and many of the traditional testing problems mentioned above can be overcome.

While the BIST approach has many positive aspects, many improvements are needed such as the following:

- *Reduce the hardware overhead of the on-chip circuitry.*
- *Reduce the test length while still obtaining a high fault coverage (98%-99%).*
- *Develop a computationally efficient procedure for the design of the required BIST hardware modules.*

Testability synthesis is an increasingly important aspect of VLSI circuit design. Time to market and other cost constraints demand that constantly smaller design teams put together circuits in ever-decreasing time. To meet the demand, successful methodologies must include design reuse and automated logic synthesis.

In the past, we could add design for testability (DfT) circuits manually, after logic synthesis. But today's need for a shorter time to market makes this an unaffordable design bottleneck. Ignoring DfT altogether can jeopardize product quality and/or introduce schedule delays. When the design process is synthesis-based, we can only achieve DfT by incorporating test and synthesis into a methodology that is as automated as possible. A thorough understanding of testability synthesis tools, then, is valuable in designing marketable circuits.

A wide variety of activities fall under the testability synthesis umbrella. These activities include, for example, synthesizing inherently testable circuits; incorporating BIST circuit in behavioral synthesis; and constraining logic synthesis to conform to the rules of standard DfT techniques like scan. There are two broad classes of test synthesis tools, based on where they fit in a typical design methodology: gate-level and register-transfer level.

Gate-level tools are the more mature of the two. They operate on a circuit that has already passed through the technology-mapping phase of logic synthesis, generating circuitry and test patterns for a chosen DfT methodology, typically full scan in today's market. Register-level tools are a more recent addition. Rather than circuitry, these tools generate RTL code that designers merge with the rest of the design before synthesis. These tools are more suited to methodologies like BIST and boundary scan, rather than full scan. and register-transfer level.

1.2 Motivation

During its life-time, a digital system is tested and diagnosed on numerous occasions. For the system to perform its intended mission with high availability, testing and diagnosis must be quick and effective. A sensible way to ensure this is to specify test as one of the system functions. In other words, self-test is the key. Digital systems involve a hierarchy of parts: modules, chips, boards, and so on. At the highest level, which may include the entire system, the operation is controlled by software. Self-test is often implemented in software. While a purely software approach to self-test may suffice at the system level, it has several disadvantages. Such testing may have poor diagnostic resolution because it must test parts designed without specific testability considerations. In addition, a good software test can be very long, slow, and expensive to develop.

In increasing attractive alternatives is built-in self-test (BIST) - that is, self-test implemented in the hardware itself. BIST is a design-for-testability (DfT) technique in which testing (test generation and test application) is accomplished through built-in hardware features. When testing is built into the hardware, it has the potential of being not only fast and efficient but also hierarchical. In other words, in a well-designed testing strategy, the same hardware can test chips, boards, and system. The cost benefits, which may not seem significant at the chip level, are enormous at the system level. Moreover, BIST offers solutions to several major testing problems.

1.2.1 The Complexity Issue

As the complexity of VLSI systems increases, we ask if the testing problem can be partitioned. The answer, unfortunately, is no. For example, consider two devices connected in a cascade. There is often no simple way to derive tests for the cascade from the given tests for its individual part. Another possibility is the use of a hierarchical approach. The complex design automation problems of synthesis and physical design are often solved through the hierarchical procedures. The testing problem, however, is not easy to solve with traditional

hierarchical techniques. For example, no simple method exists for deriving a board test from tests for chips on the board.

BIST does offer a hierarchical solution to the testing problem. Consider the testing of a chip embedded in a board that is a part of a system. The top-down hierarchy consists of system, boards, and chips. Suppose all levels of the hierarchy use BIST. To test the chip, the system sends a control signal to the board, which in turn activates self-test on the chip and passes the result back to the system. Thus, BIST provides efficient testing of the embedded components and interconnections, reducing the burden on system-level test, which need only verify the component's functionality.

1.2.2 The Quality Issue

A product's quality depends on the ability of its tests. Test ability is most frequently measured as coverage of single stuck-at faults. Thus, we calibrate tests according to their ability to detect single lines that appear as if shorted to ground (stuck-at-0) or to the power supply (stuck-at-1). Since this kind and number of faults that occur depends upon the type of devices (CMOS, bipolar, GaAs), evaluating test quality can be a complicated task. In general, quality requirements such as 95% fault coverage for complex VLSI chips or 100% coverage of all interconnect faults on a printed circuit board (PCB) are base on practical considerations. The test engineer tries to achieve a low reject ratio (percentage of faulty parts in the number passing the test) - for example, 1 in 10,000 - while controlling the test generation and test application. For very large systems, such requirements are achievable only through DfT. And, BIST is the preferred form of DfT.

1.2.3 The Test Generation and Test Application Issues

As pointed out earlier, the problem of generating tests is difficult to solve by using the hierarchy. The difficulty lies in carrying the test stimulus through many layers of circuitry to the element under test and then conveying the result again through many layers of circuitry to an observable point. BIST simplifies this problem by localizing testing.

For printed circuit board (PCB) testing, in-circuit testing is a dominating method. In this method, a bed-of-nails fixture customized for the board under test enables the testers to access the pins of the chips mounted on the board. In-circuit testing effectively applies chip tests for diagnosis and also effectively tests board wiring. The method, however, presents several problems. First, it is effective only after a board is removed from the system; therefore, it is no help in system diagnosis. Second, today's components are often mounted densely on both sides of the board. Bed-of-nails fixtures for such boards are either too expensive or impossible to build. BIST offers a superior solution to the test application problem. First, built-in test circuitry can test chips, boards, and the entire system without expensive, external automatic test equipment. Second, for off-line testing of boards and chips and for production testing, one can use the same tests and test circuitry that are used at the system level.

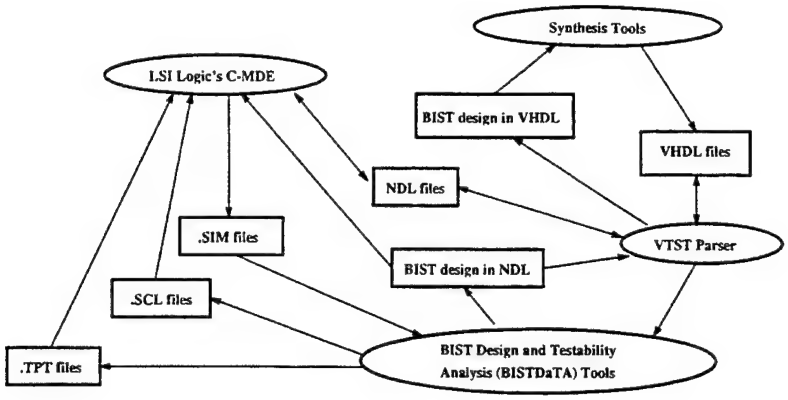


Figure 1.1: Overview of VTST system

1.2.4 The Economics Issue

At the chip level, BIST offers small savings in testing costs. But in product life-cycle costs, the savings are very in favor of BIST. It is shown that the additional expense of designing BIST hardware is somewhat balanced by the savings from test generation. Fabrication cost increases at all levels due to the extra hardware BIST requires. Testing cost decreases due to more efficient tests, less-expensive test equipment, and improved trouble-shooting during assembly and integration. Maintenance test is a system-level function involving diagnosis. This, BIST's impact on maintenance cost is greatest at the system level. BIST also reduces the diagnosis and repair costs at the board and system levels. Thus even with considerably lower benefits at chip and board levels, it is believed that BIST is still the best DfT alternative.

1.3 An Overview of VTST

VLSI Testability Synthesis Tool (VTST) is a VLSI CAD system for inserting built-in self-test (BIST) circuitry into VLSI microelectronics. It provides the user with the choice of several proven BIST methodologies, including the ability to use different methodologies within the same circuit. It provides a high level of design automation, including the automatic creation and insertion of BIST circuits into the original circuit; or the designer may manually control the design process at a lower level. It is interfaced to commercially available application-specific integrated circuit (ASIC) CAD tools including those from LSI Logic, Mentor Graphics, and Synopsys to introduce testability as an integral part of the design process. VTST provides a solution to the testability design deficiencies present in existing tools for design and synthesis of complex VLSI circuits. It includes a VHDL input and output capability and an EDIF interface capability. It currently is hosted on SUN workstations and includes a "concurrent engineering" mode that permits parallel execution of BIST tasks on networked workstations. It is designed to work with realistically sized VLSI circuits with practical tool resource utilization.

VTST is an automatic Computer-Aided Design (CAD) tool which assists VLSI designers in designing and developing ASIC's with design for test features. VTST consists of

three core CAD toolsets. They are synthesis toolset, LSI Logic's C-MDE design system, and BIST Design and Testability Analysis (BISTDaTA) toolset.

Synthesis toolset may include any commercial synthesis CAD/CAE tool that performs the logic synthesis task. BISTDaTA is a BIST design generator and testability analysis tool which serves as a testability advisor as well as a BIST design generator to produce a final design adding the design for testability features. A structural description written in VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) is the interfacing medium between commercial synthesis CAD/CAE tools and BISTDaTA. As shown in Figure 1.1, VTST Parser serves as a front end interfacing tool between commercial CAD/CAE tools and BISTDaTA. VTST Parser takes in the VHDL structural description generated from synthesis tools and feeds its output files to BISTDaTA. Also, VTST Parser provides a path to LSI Logic's C-MDE system for users to perform logic analysis, placement and routing.

After BISTDaTA, A BIST design of the desired circuit under test (CUT) in LSI Logic's Netlist Description Language (NDL) codes will be generated with all necessary BIST hardware inserted. Users may employ the VTST Parser to convert NDL back to VHDL. On the other hand, VTST utilizes LSI Logic's C-MDE system via netlist translation to verify gate-level as well as physical level performance after the BIST hardware insertion, and also verify the test and normal operations. Both of **.SCL** and **.TPT** files are generated by BISTDaTA to provide simulation control files for simulating BIST design in the C-MDE environment. C-MDE will generate a simulation result **.SIM** which is an input file to BISTDaTA for a validity check of the final BIST design.

1.4 Scope

In this report, an architecture for automated BIST design that VTST can generate is first introduced in Chapter 2. Then, the VTST system will be presented in the following order.

- An interfacing parser, namely VTST Parser, will be introduced in Chapter 3.
- An overview of BIST Design and Testability Analysis tool (BISTDaTA) will be described in Chapter 4.
- A circuit preprocessing tool: Sensor-Scissor will be described in Chapter 5.
- A pseudo-scan processing tool: Pseudo-Scan Scissor will be described in Chapter 6.
- An automated synthesis tool for pseudo-exhaustive test generator: BISTSYN will be described in Chapter 7.
- A circuit partitioning tool - Autonomous will be described in Chapter 8.
- A BIST design tool for sequential circuit - SEQBIST wiil be described in Chapter 9.

- A random memory testing scheme will be described in Chapter 10.
- An NDL BIST automation tool - BUILDER will be described in Chapter 11.
- VTST test scheme for automated BIST design will be described in Chapter 12.
- VTST distributed system will be described in Chapter 13.
- VTST graphics user interface will be described in Chapter 14.

Finally, the results of two test cases: the system control coprocessor (CP0) of the LR33000 CPU core and the Enhanced Memory Chip (EMC) will be discussed in the Chapters 15 and 16.

Chapter 2

Architecture for Automated BIST Design

Design for Testability (DfT) is in great demand as the density and the number of devices increases in today VLSI chips. Manufacturing defects are undesirable for chip makers and chip malfunctions may cause real time systems to go down and/or cause tragedy. Fault-free chips and ensuring fault-free VLSI chips in real time systems are highly in demand.

Built-In Self-Test (BIST) provides a very effective method to ensure that VLSI chips in real time systems are fault-free. Without external automatic test equipment on site, BIST design VLSI chip is capable of performing self-test and providing the status of the VLSI chip. With simple and few control test signals required, the users or real-time systems/application engineers will be able to ensure the status of the VLSI chip in a short period of time.

The basic BIST architecture requires the addition of three hardware blocks to a digital circuit: a test pattern generator, a test response analyzer, and a test controller. Examples of test pattern generators are a ROM with stored patterns, a counter, and a linear feedback shift register (LFSR). A typical test response analyzer is a comparator with stored responses or an LFSR used as a signature analyzer. A control block is necessary to activate the test and analyze the responses. In general, test functions can be executed through a test manager (test controller) circuit.

Consider a hierarchical application of the BIST concept. The system consists of several circuit boards. Each board may contain several VLSI chips. The test manager at the system level can simultaneously activate self-test on all boards. The test manager on each board, in turn, activates self-test on each chip on that board. A chip test manager is responsible for executing self-test on the chip and then transmitting the results (fault-free or faulty) to the system test manager. Using these results, the system test manager can isolate the faulty chips and boards.

The effectiveness of this diagnosis procedure depends on the self-test implemented on the chips. Thus, fault coverage is a major issue in BIST designs. Other important issues

are hardware overhead, additional pins required for test and performance penalty.

At the chip level, BIST involves the application of test patterns to the logic to be tested and observation of the corresponding responses. Often, the test engineer modifies on-chip logic, using some DfT technique such as scan, so that latches and flips-flops can be controlled independently of the circuit's combinational logic. However, logic may intervene between the test pattern generator and the CUT and between the CUT and the response analyzer.

We now discuss BIST test pattern types, the means of obtaining them, and related fault coverage issues. Distinct BIST methodologies are associated with each type of test pattern.

Stored-pattern BIST may use programs or microprograms, typically stored in ROM, to perform functional tests of the hardware. Successful applications of such techniques exist, but they are not our focus here. In alternative techniques, we use traditional automatic test pattern generation (ATPG) and fault simulation to generate the test patterns. We store the patterns on the chip or board, apply them to the CUT when BIST is activated, and compare the CUT responses with the corresponding stored responses. Because of the stored data's magnitude, this method is attractive only in limited cases.

Exhaustive-pattern BIST eliminates the test generation process and has very high fault coverage. To test an n-input block of combinational logic, we apply all possible 2^n-input patterns to the block. It will make the exhaustive-pattern BIST impractical for a circuit with n greater than about 25. Thus, we must partition or segment the logic into smaller, possibly overlapping blocks with fewer than n inputs. Then, we exhaustively test each block. This approach is called pseudo-exhaustive pattern BIST. Fault coverage for the exhaustive or pseudo-exhaustive method is nearly 100% and, with proper design, can be achieved without fault simulation.

In contrast with other methods, pseudo-random pattern BIST may requires a long test time and evaluation of fault coverage by fault simulation. This method, however, has the potential for lower hardware overhead and less design effort than the proceeding methods. In pseudo-random test patterns, each bit has an approximately equal probability of being a 0 or a 1. The number of patterns applied is typically of the order of 10^3 to 10^8 and is related to the circuit's testability and the fault coverage required.

Clearly when we apply test patterns to test the CUT, we must know its fault-free responses. For a given set of test vectors applied in a particular order, we can obtain the expected responses and their order from a known-good CUT or by simulating the CUT. Similar to stored-pattern BIST, we can also store responses in on-chip ROM, but such a scheme can require too much silicon area to be of practical value. Alternatively, methods that compress the test pattern and the corresponding responses of a fault-free CUT and regenerate them during self-test are also of limited value for general VLSI circuits.

The signature analysis function was first used by Hewlett-Packard as a compaction function. In this method, the response sequence is fed to an LFSR. The compacted sequence,

whose length is the same as that of the LFSR, is called the signature of the CUT for the applied test vector sequence. We can also use a space compactor, typically a linear circuit, to reduce the number of outputs to be compacted. For the signature method, we integrate the linear circuit with the LFSR to obtain a multiple-input linear feedback shift register (MISR), which compacts the output sequences from a multiple-output CUT. An MISR can be viewed as performing space compaction while compacting the output sequences from a CUT.

By far, the most popular compaction function is signature analysis, realized by means of an LFSR or an MISR. These structures are easy to implement, and because they are serially scannable, they can be read out easily by an external tester at the completion of self-test.

As the number of deterministic test patterns increases, the size of the RAM/ROM will increase. It is not desirable to store a large number of deterministic test patterns on chip. It may be the case that the deterministic test patterns are not easily generated from LFSR for particular hard-to-test faults present in CUT. However, these hard-to-test faults present in the CUT are able to be detected by applying different BIST methodologies to increase the controllability and observability of internal nodes. Therefore, our focus is to present a BIST architecture that not only can be automatically synthesized by the CAD tools but also provide a systematic test approach to perform BIST design of VLSI circuits. This architecture is based on the methodology that all CUTs share the one TPG and one MISR. We use LFSR designs for both TPGs and MISRs with minor variations. Since chip silicon area is a concern, sharing BIST test hardware among CUTs is desirable given the limitation of the additional routing involved. The BIST design produced by VTST is a specific defined architecture. With the architecture, self-test is performed within the chip. Fault-free or faulty signal will be generated through an output pin.

2.1 BIST Architecture

The BIST architecture aims at providing a few external signals via input pins to initiate the self-test within the chip and the status of the chip will be reported by one output pin. The BIST architecture introduced in this chapter is depicted in Figure 2.1. It basically consists of a Stimuli Generation (SG) module, a Response Analysis and Compression (RA&C) module, and an Internal Status Comparison (ISC) module, a Fault Status Generator (FSG), and system multiplexers. To minimize the silicon area utilization, SG, RA&C, and ISC modules can be shared among CUTs by insertng the Mux's to direct test patterns/stimuli generated from TPG residing in SG module to the desired CUT as well as direct the responses from the desired CUT to the MISR residing in the RA&C module.

2.2 Stimuli Generation Module

The Stimuli Generation (SG) module is a test stimuli/pattern generator and also is a pattern generation halt module. As shown in Figure 2.2, it consists of a self-terminating

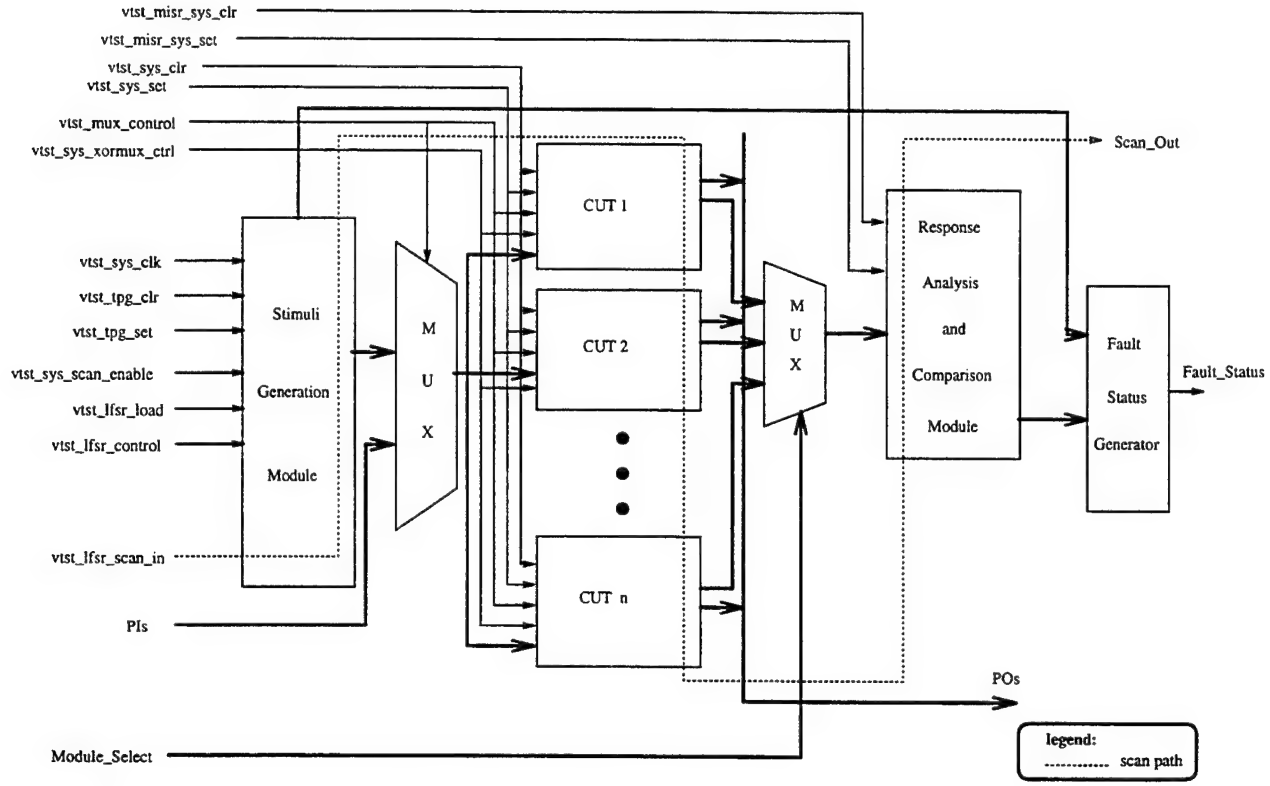


Figure 2.1: BIST architecture

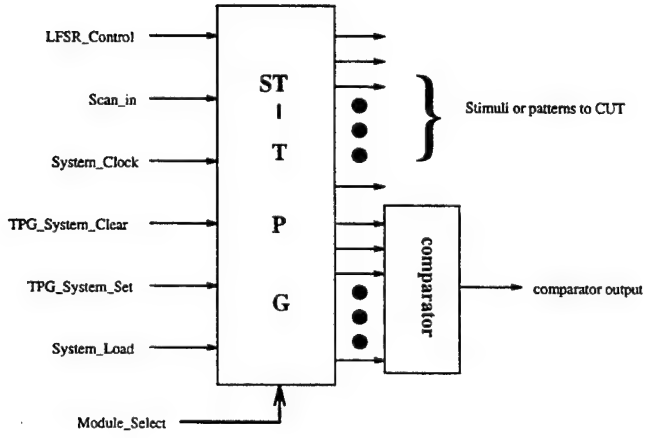


Figure 2.2: Self terminating test pattern generator

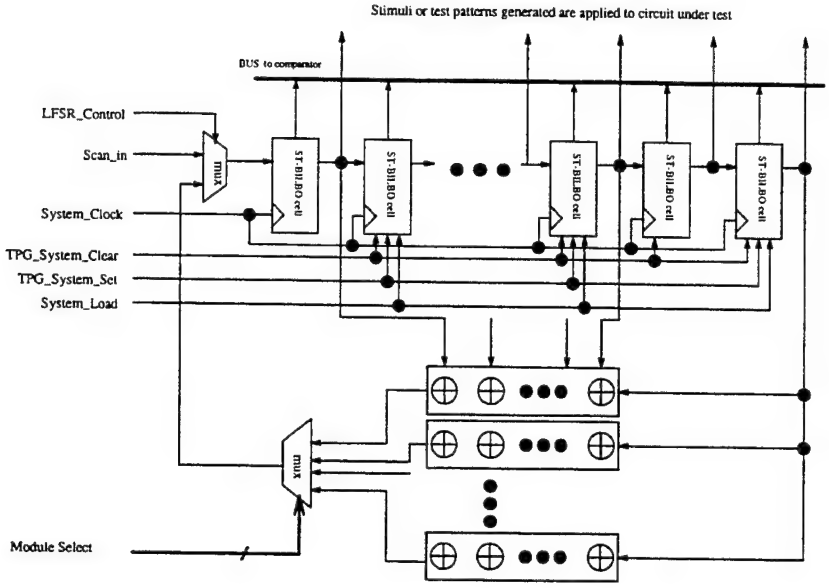


Figure 2.3: Self terminating test pattern generator (ST-TPG)

test pattern generator (ST-TPG) and a comparator which is simply an AND-tree. The ST-TPG is not just a conventional TPG, moreover, it will compare the current generated pattern with the pre-stored terminating seed (pattern). If these two patterns are matched then it will acknowledge a signal to the system to halt the pattern generation. In other words, the pattern generation for the desired CUT is complete. This is accomplished by the comparator where the ouput is pulled 'HIGH' if the generated test pattern matches the stored seed. Otherwise, it stays LOW.

If the pattern generated is not matched then TPG will continue to generate the next test pattern. Figure 2.3 depicts the architecture of a generic ST-TPG. Since the test hardware proposed for BIST design will be shared among all the CUTs on chip, the ST-TPG is required to generate test patterns with different bit sizes according to the size of the primary inputs of the selected CUT. Therefore, a series of XOR gates converge the required taps from ST-TPG to form different size of the LFSR. The taps that fed to the XOR gates are dependent upon the primitive polynomial of the LFSR. A *Module_Select* control signal directs the Mux to select the desired size of the LFSR. The ST-TPG contains a sequence of self-terminating Built-In Logic Block Observation (BILBO) cells which is depicted in Figure 2.4 where a latch is added to the conventional BILBO to store the terminating seed and is enabled by *vtst_lfsr_load*. The XNOR gate is used to perform comparison of QN port output of the D-type flip-flop and the QN port output of the latch. Note that ST-TPG is part of the scan chain for full and partial scan modes.

2.3 Response Analysis and Compression (RA&C) Module

The responses from the CUT after stimuli are applied are the only clue to verify if the CUT has faults or not. MISR has been chosen to accumulate and compact the responses to

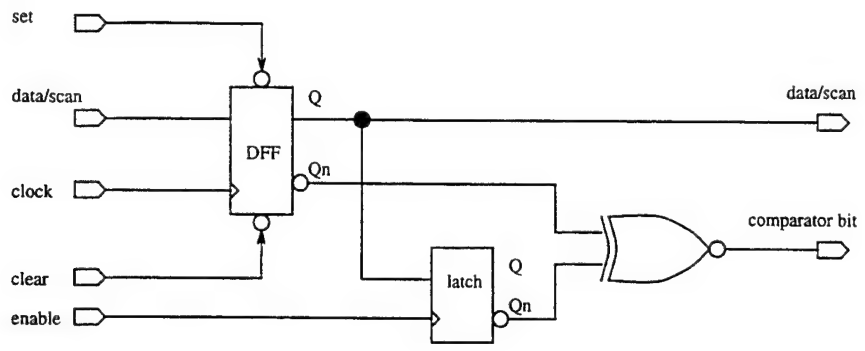


Figure 2.4: Self terminating BILBO cell

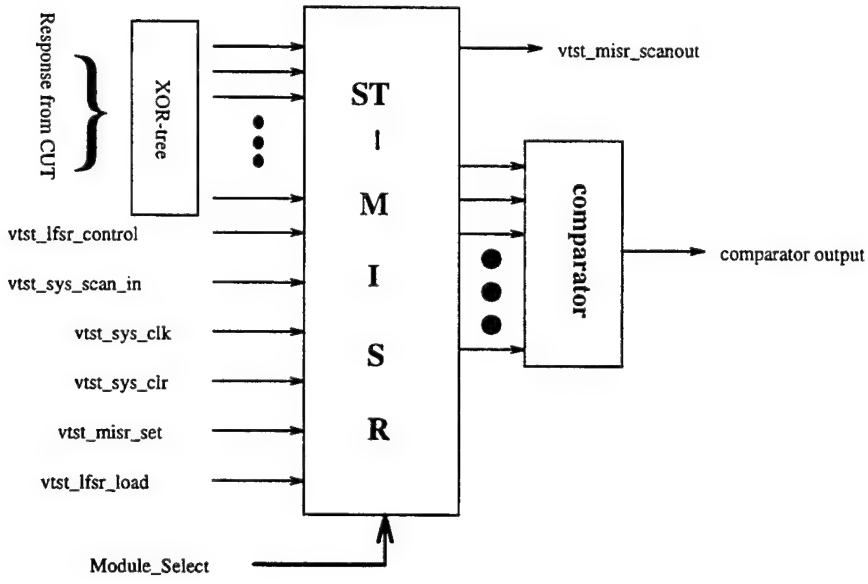


Figure 2.5: Response Analysis and Compression (RA&C) module

generate a final signature for verification. A conventional MISR is not able to pre-store the known-good signature and compare it with the generated signature after the test pattern is applied. In the proposed BIST architecture, we have designed a Response Analysis and Compression (RA&C) module, to store the known-good final signature and compare each compaction result (signature). Figure 2.5 depicts its block diagram. The RA&C consists of a self-testing MISR (ST-MISR), an XOR tree (a tree of 2-input XOR gates which converges the output responses of the CUT to no more than 64 bits), and a comparator which is a simple AND-tree. The ST-MISR is shown in the Figure 2.6, which is capable of storing the known-good signature for signature comparison. It has a similar architecture as ST-TPG except the ST-BILBO is replaced with ST-MISR-CELL. Figure 2.7 depicts the internal structure of ST-MISR-CELL. Note that ST-MISR is part of the scan chain for full and partial scan modes.

2.4 System Control Signals and BIST I/O pins

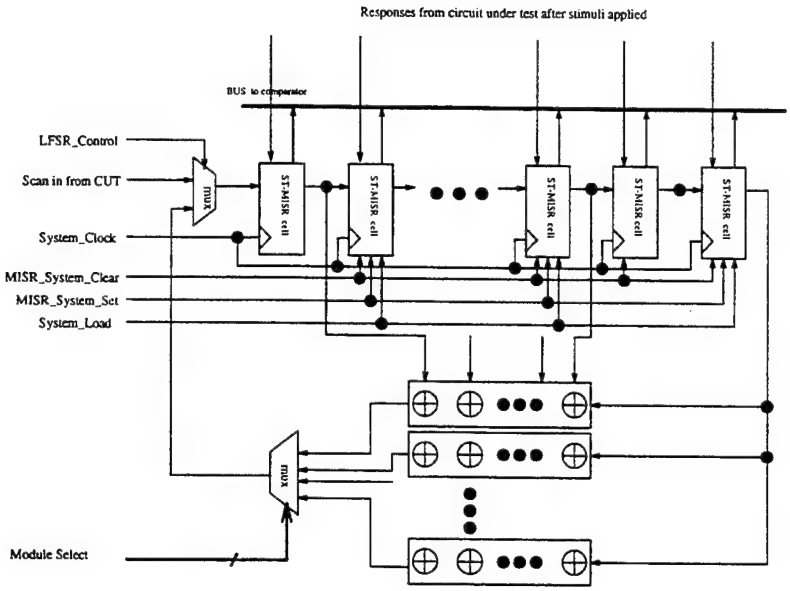


Figure 2.6: Self terminating MISR (ST-MISR)

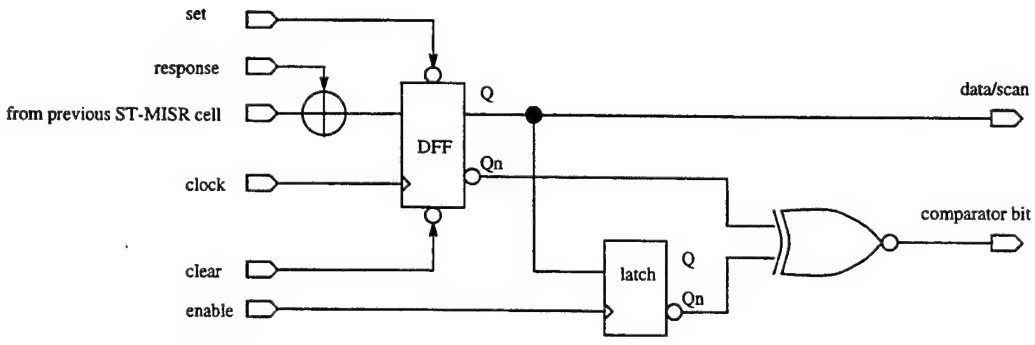


Figure 2.7: Self terminating MISR Cell

There are fourteen system control signals which can be externally excited by Automatic Test Equipment (ATE) or internally excited by an on-chip BIST controller to perform the self-test operation within the chip. These signals are:

- *vtst_sys_clk:* provides a system clock for both test and normal operations.
- *vtst_mux_ctrl:* controls all the Mux's to select test or normal operation. During test mode, *vtst_mux_ctrl* should be pulled 'HIGH' all the time. Otherwise, it stays 'LOW'.
- *vtst_sys_clr:* is 'LOW' activated. The signal *vtst_sys_clr* clears the contents of all internal flip-flops and latches except those flip-flops that are selected as scan flip-flops or latches residing in a scan chain. The *vtst_sys_clr* should stay 'HIGH' to disable the *clear* operation of flip-flops or latches. In order to clear the contents of all flip-flops and latches, the signal *vtst_sys_clr* should be pulled "LOW" to enable the *clear* operation of flip-flops and latches right before the very first test pattern is applied to the circuit under test.
- *vtst_sys_set:* is 'LOW' activated. The signal *vtst_sys_set* sets the contents of all internal flip-flops and latches except those flip-flops that are selected as scan flip-flops or latches residing in a scan chain. To set the contents of all flip-flops and latches, the signal *vtst_sys_set* stays 'HIGH' to disable the *set* operation of all the chained flip-flops and latches. The *vtst_sys_set* should be pulled 'LOW' to enable the *set* operation of all the flip-flops and latches right before the very first test pattern is applied to the circuit under test.
- *vtst_tpg_clr:* is 'LOW' activated. The signal *vtst_tpg_clr* clears the contents of the TPG's flip-flops. Since the TPG is physically connected to primary inputs (PIs) of the CUT via Mux's, undesired signals may be applied to CUT during the scan/shift mode of the TPG. By pulling *vtst_tpg_clr* signal 'HIGH' for one clock cycle before the next seed is scanned in, undesired signals applied to the CUT during shift mode of the TPG can be avoided. During the very first one or two clock cycles, *vtst_tpg_clr* is pulled 'LOW' to enable the *clear* operation of flip-flops residing in the TPG. After that, it can stay 'HIGH' throughout the test mode.
- *vtst_tpg_set:* is 'LOW' activated. To have all 1's pattern as an initial seed, scanning in all 1's pattern is not necessary. The signal *vtst_tpg_set* can be pulled 'LOW' to set the contents of the TPG to all 1's so that the initial seed (all 1's) can reside in the TPG in one clock cycle. To do that, *vtst_tpg_set* is pulled 'LOW' right after the terminating seed and final known-good signature have been loaded to the TPG's and the MISR's latches respectively. Then, *vtst_tpg_set* is pulled 'HIGH'.
- *vtst_misr_clr:* is 'LOW' activated. The signal *vtst_misr_clr* clears the contents of the MISR before the signature generation process is started. The signal *vtst_misr_clr* must be pulled 'LOW' to enable the *clear* operation of flip-flops residing in the MISR during the very first test pattern applied. Then, the signal *vtst_misr_clr* should stay 'HIGH'.
- *vtst_misr_set:* is 'LOW' activated. The signal sets the contents of the MISR before the signature generation process is started. The signal *vtst_misr_set* must be pulled

'LOW' to enable the *set* operation of flip-flops residing in the MISR during the very first test pattern is applied. Then, the signal *vtst_misr_set* should stay 'HIGH'.

- *vtst_lfsr_load:* is 'HIGH' activated. It will enable the latches residing in both the TPG and the MISR to load the terminating seed and final known-good signature to TPG and MISR respectively. This signal should stay 'LOW' all the time except when the loading seed/ signature operation is desired. This operation must be achieved when the terminating seed and final signature have been scanned/shifted to the desired positions of the TPG and MISR.

- *vtst_lfsr_control:* is a signal which will distinguish the scan/shift operation from the LFSR mode of both the TPG and the MISR. When the signal *vtst_lfsr_control* stay 'HIGH', then TPG and the MISR will perform LFSR mode. While TPG and MISR perform scan/shift operation, it stays 'LOW'. It is desirable to pull *vtst_lfsr_control* 'LOW' to scan in the seeds and final signature. When the last bit of initial seed is scanned in, *vtst_lfsr_control* is pulled 'HIGH'.

- *vtst_sys_scan_enable:* is employed only for scan design. It is a scan control signal controlling all scan flip-flops and latches.

- *vtst_lfsr_scan_in:* is the input pin for external scan purpose. In non-scan test, initial seed and terminating seed for TPG, and final signature for MISR are scanned in through this pin. In scan design, it is the pin for scanning in the scan patterns.

- *vtst_sys_xormux_ctrl:* is the control pin for directing CBIST Mux to perform test or normal operation. When it is 'HIGH', it performs test operation. Otherwise, it performs normal operation.

- *Module_Select_Pins:* are a set of control signals/pins to select the desired CUT within the chip for testing. If there are 3 or 4 CUTs within a chip, 2 module select pins are required and they are denoted as *vtst_module_sel_0* and *vtst_module_sel_1*.

2.5 Fault Status Generator

Fault Status Generator (FSG) is a two-gate structure module which consists of a 3-input AND gate and a D-type flip-flop. The D-type flip-flop is used to delay the comparator output bit of SG module for one clock cycle so that the comparator output bit of RA&C module and the comparator output bit of SG can arrive at fault status generator at the same time. The SG comparator output bit is pulled 'high' when the test generation is complete and the RA&C comparator output bit is pulled 'high' when the signature matches the known-good one. If this is the case, then the output for FSG will be pulled 'HIGH' indicating that the CUT is fault-free.

2.6 System Multiplexers

There are two major groups of Mux's used in the BIST architecture. They are the group of Mux's between the SG module and the CUT and the group of Mux's between the CUT

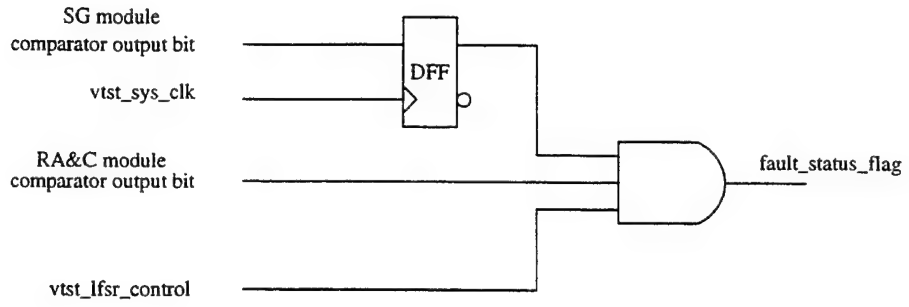


Figure 2.8: Fault Status Generator (FSG)

and the RA&C module. These Mux's are controlled by the external *Module_Select_Pins* to direct the generated test patterns to desired the CUT and direct the responses from the desired CUT to RA&C module. These embedded Mux's are controlled by the system control signal *vtst_mux_control*.

Chapter 3

VTST Front-End Parser

VTST is designed to explore the design space with testability constraints included during design and synthesis processes. It provides interface software to communicate with the commercial CAD/CAE design tools such as Synopsys, Mentor Graphics, Compass, and LSI Logic Inc. CMDE system. Since VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) has become one of the standard industrial *hardware description languages (HDL)*, and LSI Logic Inc. Netlist Description Language (NDL) has been a commonly used circuit format in ASIC design, VTST provides a front-end parser which parses the input circuit generated by these CAD/CAE tools in either format (VHDL or NDL) and generates an intermediate description netlist for BIST design and testability analysis. As shown in Figure 3.1, VHDL and NDL are the interfacing HDLs that these commercial CAD/CAE tools and VTST use.

The front-end parser that performs the parsing task is named *VTST Parser*. Figure 3.2 depicts the parsing process done by VTST Parser. There are four system files that are required for VTST Parser to proceed with the parsing process. They are denoted as *.file, .inlib, .outlib, .ucomplist.* Each of these system files plays a role in the parsing process.

The system file, namely *.file*, contains all the file names of subcircuits of the CUT. The system files *.inlib* and *.outlib* contain file names of input and output technology library map-

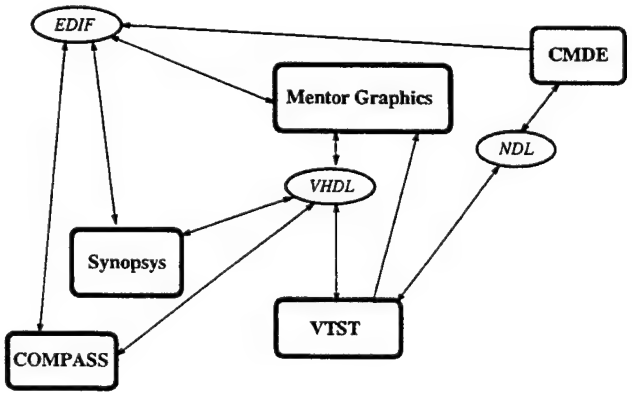


Figure 3.1: VTST interface with commercial CAD/CAE tools

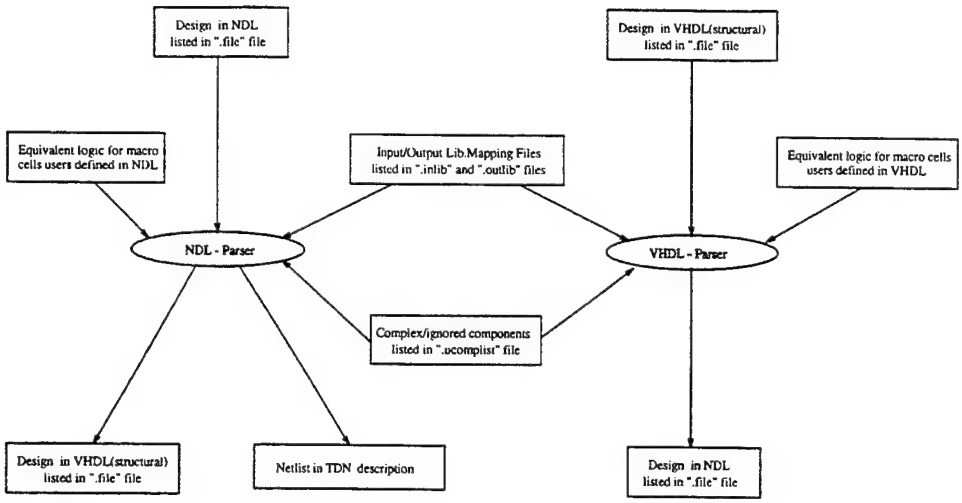


Figure 3.2: VTST parsing stage

ping files respectively, which explicitly describes the functionality of each primitive logic element such as *AND* gates, *OR* gates. Finally, the file *.ucomplist* lists all the components that are not described in the netlist or those components that are not easily defined such as RAM or ROM. VTST Parser will read in these components listed in the *.ucomplist* file and treat them as primitive elements like macro cells. In addition, for testability analysis VTST Parser requests users to define the equivalent logic for all macro cells used in the design in order to flatten the macro cells to structures that consists of only primitive logic elements.

With these system files and the user-defined equivalent logic of all the macro cells, VTST Parser will perform the parsing process to generate the Testability Description Netlist (TDN) for BIST design and testability analysis. The process is described in Figure 3.2.

In this chapter, Section 3.1 will discuss how to prepare the four system files to be used during the parsing process. Section 3.2 will describe the preparation of equivalent logic for the macro cells. Section 3.3 will present an example of the TDN format. Sections 3.5 and 3.4 will describe the VTST Parser for VHDL and NDL.

3.1 VTST System Files

This section describes the four system files required by the VTST Parser to perform the parsing process.

3.1.1 Design Files - *.file*

A design input to the VTST system is most likely described using the top-down hierarchical approach. It is unlikely that all modules/components in the design are written in one single file. They may be described in different files and these files may also reside in different directories. VTST allows users to include all files that are associated with the design but they may reside in different files and in different directories. To read all these

files, a system file called *.file* is used to keep track of the full path and file names of all these files. VTST provides the user interface to create *.file* for system use (see Chapter 14).

3.1.2 Library Technology Files - *.inlib, .outlib*

VTST is a technology library independent tool. It is unlikely to develop a parser that is capable of recognizing primitive logic elements for all the libraries being used in current ASIC markets. For instance, *an2* is the name given to a 2-input *AND* gate for LSI Logic Inc's LCA200K library. On the other hand, *an02d1* is the name given also to a 2-input *AND* gate for VLSI Technology Inc. VSC450 library. In order to enable the VTST Parser to understand/recognize the primitive logic elements, a look-up table must be built by the users for the VTST Parser to perform the table search. The file *.inlib* is a system file that the VTST Parser will look for to find where the look-up table resides. The technology library mapping file should reside in this directory $VTST_HOME/vtstlib/tech directory where $VTST_HOME is the directory in which VTST system is installed. However, the users may define the library technology mapping file and put it in the current working directory, then the full path of the library technology mapping file must be stated in the system file *.inlib*. If there is no library used, -1 should be stated in the file *.inlib*.

The look-up table (library technology mapping file) is a table in which all the primitive logic elements are specified. The file begins with the name of the cell library or gate array technology as shown in the first line of the file in Figure 3.3. The table includes not only the names of primitive logic elements but also the detailed information for parsing purpose. For any primitive logic element, other than flip-flops and latches, which have more than one output port, the information listed in the technology library mapping file must be in the format as shown in the following order.

1. The name for a gate according to the library.
2. It is a basic gate or complex component. If it is a basic gate, alphabet "b" is given in this field. Otherwise, alphabet "c" is given.
3. The function name of the gate must be given in this field. The function name used must be as stated as follows: **AND** for AND gate, **OR** for OR gate, **NAND** for NAND gate, **NOR** for NOR gate, **XOR** for XOR gate, **XNOR** for XNOR gate **INV** for inverter and **BUFF** for buffer.
4. The gate count or area as stated in library data book.
5. The number of input ports.
6. The name of each input port as stated in the library data book such as **a,b,c etc**.
7. The name of the output port as stated in library data book such as **z** or **zn**.

For instance, a 2-input *AND* gate and a 3-input *OR* gate of LSI Logic Inc's LCA300K library are described in the technology library mapping file as shown in Figure 3.3. The name of the LCA300K 2-input *AND* gate is *an2*, it is a basic gate, and its function name is *and* and its gate count is 4 and the number of inputs is 2. The first input port is *a*, the second input port is *b* and the output port is *z*.

VTST TECH LIBRARY : lca300k

an2	*b*	*and*	4	2	*a*	*b*	*z*					
or3	*b*	*or*	4	3	*a*	*b*	*c*	*z*				
fd2a	*b*	*dff*	21	*d*	−1	*cp*	*q*	*qn*	*cd*	−1	−1	*r*
ld1a	*b*	*latch*	13	*d*	−1	*cp*	*q*	*qn*	−1	−1	−1	*h*

Figure 3.3: Example of library technology mapping file

One the other hand, for flip-flops and latches which have more than one output port, the information listed in the technology library mapping file must be in the format in the following order.

1. The name for a gate according to the library.
2. It is a basic gate or complex component. If it is a basic gate, alphabet "b" is given in this field. Otherwise, alphabet "c" is given.
3. Function name of the gate must be given in this field. The function name used must be as stated as follows: **dff** for a D flip-flop and **latch** for a D-type latch.
4. The gate count or area according to the library databook.
5. The name of the first input data port such as D port of D flip-flop, and J port of JK flip-flop.
6. The name of the second input data port such as K port of JK flip-flop, but -1 for D flip-flop.
7. The name of the clock input such as CP or CPN.
8. The name of the first output port such as Q.
9. The name of the second output port such as QN.
10. The name of the clear input such as CD or -1 if the clear input doesn't exist.
11. The name of the set input such as SD or -1 if the set input doesn't exist.
12. The name of the test input such as TI or -1 if the test input doesn't exist.
13. The name of the test enable input such as TE or -1 if the test-enable input doesn't exist.
14. The edge sensitive indicator: 'r' for the rising edge trigger; 'f' for the falling edge trigger; 'h' for the high level sensitive; or 'l' for the low level sensitive.

For instance, a D-type flip-flop with clear and a D-type latch of LSI Logic Inc's LCB007 library will be described as follows in the technology library mapping file: Also VTST offers an option for the users to generate the same design but choosing a different technology library. This target library is the output technology library users specified in ".outlib" file. The advantage of this is to allow the BIST design to be fabricated with a desired technology.

3.1.3 Unknown/Ignored Components

Since some components in the input netlist may not be easily defined using primitive logic gates and some of the components such as ROM and RAM may need to have individual testability analysis performed, these components will be listed in a system file called *.ucomplist* file. VTST Parser will not search for those components in the technology library or all the design files. It will treat them as primitive elements and the testability analysis will be done individually.

3.2 Equivalent Logic Cell library

The *primitive* element can be defined as an element which can't be expanded further to a lower level of description in a particular design environment. For instance, a 2-to-1 multiplexer in LCA200K library of LSI Logic Inc, is a primitive element in the C-MDE environment. However, the testability analysis may not be performed on this primitive element because stuck-at faults may exist within this primitive element. To perform testability analysis, this primitive element can be replaced with equivalent logic such as two 2-input AND gates and 2-input NOR gates.

Most VLSI design tools are developed to support cell based design. The macro cell library developed for ASIC has a smaller silicon area than the cell basic design. All the macro cell libraries developed by the commercial CAD/CAE vendors are primitive as described in the previous paragraph. To perform testability analysis, these macro cells have to be replaced with the primitive gates such as AND gates, OR gates, etc.

The VTST Parser is capable of detecting all components which are not defined in input circuit files specified in *.file*, in the technology library mapping file specified in *.inlib*, in the *.ucomplist* file. These undefined components are required to be defined either in VHDL if the input circuit is coded in VHDL, or otherwise, in NDL.

We have developed the equivalent logic cell libraries in NDL for LSI Logic LCB007, LCA100K, LCA200K, LCA300K and the equivalent logic cell library in VHDL for COMPASS VSC450 library. They can be found in $VTST_HOME/vtstlib/cellib directory. Let's use an OR-AND macro cell as an example. OR-AND is a macro cell that has two 2-input OR gates and one 2-input NAND gate. Figure 3.4 depicts its schematic diagram of this macro cell. The oa01d1 is the name given in COMPASS library. For this macro cell, the equivalent logic defined in VHDL and in NDL is shown in the following examples.

Example 1: oa01d1 in NDL netlist:

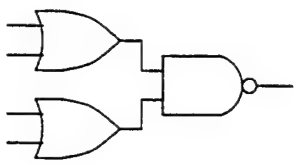


Figure 3.4: Equivalent logic for OR-AND gate

```
compile;
module oa01d1;
inputs a1,a2,b1,b2;
outputs zn;
define
u0(u0_out=z) = or02d1(a1=a1,a2=a2);
u1(u1_out=z) = or02d1(b1=a1,b2=a2);
u2(zn=z) = nd02d1 (u0_out=a1,u1_out=a2);
end module;
end compile;
```

Example 2: oa01d1 in VHDL netlist:

```
library work;
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity oa01d1 is
port (a1,a2,b1,b2 : in std_logic; zn : buffer std_logic);
end oa01d1;
architecture behavior of oa01d1 is
component nd02d1
port (a1,a2 : in std_logic; zn : buffer std_logic);
end component;
for all : nd02d1 use entity work.nd02d1(behavior);
component or02d1
port (a1,a2 : in std_logic; z : buffer std_logic);
end component;
for all : or02d1 use entity work.or02d1(behavior);
signal u0_buffer,u1_buffer: std_logic;
begin
u0: or02d1 port map(a1,a2,u0_buffer);
u1: or02d1 port map(b1,b2,u1_buffer);
u2: nd02d1 port map(u0_buffer,u1_buffer,zn);
end behavior;
```

3.3 Testability Description Netlist (TDN)

Testability Description Netlist (TDN) is an intermediate format which is designed for testability use in VTST. Unlike most of the HDLs, TDN doesn't support a hierarchical top-down design approach. A design described in TDN is a flattened netlist description. This TDN file contains all the components of the design.

3.3.1 TDN File Structure

A TDN file basically consists of three mandatory portions and an optional portion. The mandatory portions are listed in the TDN file in the following order: the *circuit declaration section*, the *input/output declaration section*, and the *connection section*. The optional portion is named as the *direction connection section*.

The *circuit declaration section* is the very first section presented in the TDN file. It starts with the name of the circuit and then the characteristics of the circuit are introduced. The *input/output declaration section* specifies input/output pins of the design. The connection section explicitly describes the interconnections among inputs, outputs, and gates. The direction connection section describes those inputs which are directly connected to the outputs.

3.3.2 Circuit Declaration Section

The *circuit declaration section* displays characteristics of the design/circuit in the following order:

1. the name of the design,
2. the number of inputs,
3. the number of outputs,
4. the number of flip-flops,
5. the number of tristate buffers,
6. the number of inverters,
7. the number of other logic gates
8. the number of ignored components.

The following rules and syntax are introduced to guide VTST users to write a TDN file. We use **BOLD** print to represent the reserved words, and *italic* print to represent users inputs.

Each item listed below must start with a '#' sign and then followed by either the circuit information or the reserved words. There are two ways of creating syntax for the *Circuit Declaration Section*. The first syntax is listed as follows:

1. # *circuit_name*
2. # *number_of_inputs* **inputs**
3. # *number_of_outputs* **outputs**
4. # *number_of_d-type_flip-flops* **D-type flip-flops**
5. # *number_of_jk-type_flip-flops* **JK-type flip-flops**
6. # *number_of_t-type_flip-flops* **T-type flip-flops**
7. # *number_of_latches* **latches**
8. # *number_of_inverters* **inverters**
9. # *number_of_tristate_buffers* **tristate buffers**
10. # *number_of_ignored_components* **ignored**
11. # *number_of_total_gates* **gates** (*number_of_and_gates* **ANDs** + *number_of_nand_gates* **NANDs** + *number_of_or_gates* **ORs** + *number_of_nor_gates* **NORs** + *number_of_xor_gates* **XORs** + *number_of_xnor_gates* **XNORs** + *number_of_buffers* **BUFFs**)

The second way of writing the Circuit Declaration Section is:

1. # **Circuit:** *circuit_name*
2. # **Inputs:** *number_of_inputs*
3. # **Pseudo_PIs:** *number_of_pseudo-inputs*
4. # **Outputs:** *number_of_outputs*
5. # **Pseudo_POs:** *number_of_pseudo-outputs*
6. # **DFF:** *number_of_d-type_flip-flops*
7. # **JKFF:** *number_of_jk-type_flip-flops*
8. # **TFF:** *number_of_t-type_flip-flops*
9. # **LATCH:** *number_of_latches*
10. # **Tristates:** *number_of_tristate_buffers*
11. # **gates:** *number_of_total_gates*
12. # **Ignored:** *numbe_of_ignored_components*
13. # **Direct Connection:** *number_of_direct_connect*

3.3.3 Input/Output Declaration Section

The *input/output declaration section* is the declaration section for defining all the inputs, outputs, pseudo-inputs, pseudo-outputs, and any scan inputs and scan outputs. In this section, the reserved words used are only the following ones.

1. **Inputs(** *(name_of_input_port)*
2. **Outputs(** *(name_of_output_port)*
3. **Pseudo_PI(** *(name_of_Pseudo_input_port)*
4. **Pseudo_PO(** *(name_of_Pseudo_output_port)*
5. **Scan_PI(** *(name_of_Scan_input_port)*
6. **Scan_PO(** *(name_of_Scan_output_port)*
7. **CTRL(** *(name_of_control_signals)*

3.3.4 Interconnection Section

Interconnection Section explicitly describes the interconnections among primary inputs, logic gates and primary outputs. To instantiate a component or gate, the following order must be followed.

1. the name of the component/gate,
2. output connection,
3. '='
4. the function name of component/gate,
5. input connection.

As shown in the following,

u25_1##u0_0(l_54=z)=or02d1(l_3=a1, l_4=a2)

describes an instantiation of a component or02d1 with all the input and output interconnections. The string "u25_1##u0_0" is the name of the gate, (l_54=z) is the output connection of the instant u25_1##u0_0, where l_54 is the signal and z is the output port name of the gate or02d1. It is shown that or02d1 is the function name of the component/gate, and (l_3=a1, l_4=a2) is the input connection of 25_1##u0_0 where l_3 and l_4 are the signals and a1 and a2 are the input port names of the gate or02d1. The name of the component "25_1##u0_0" indicates that u0_0 is inside the component 25_1 and ## separates the hierarchical levels.

3.3.5 Optional Direct Connection Section

Direct connection is optional. If there is any direct connection in the VHDL/NDL code, TDN will be generated with the syntax:

$$name\_of\_input\_port \Rightarrow name\_of\_output\_port$$

3.3.6 Reserved Words

There are several reserved words used in TDN which helps the VTST Parser to understand the nature/characteristics of the circuit under test. These reserved words can only be used in the *circuit declaration section* and *input/output declaration section* of a TDN file. Figure 3.5 depicts a sample of TDN.

3.4 NDL Parser

The NDL Parser is one of the key front-end parsers developed in VTST. Its primary task is to translate the hierarchical design of a CUT into a TDN description for testability analysis. Also, the NDL Parser works as a translator to convert the NDL description to a VHDL structural description for interfacing with the commercial CAD/CAE tools.

The key features of the NDL Parser include the following.

1. It is a technology independent parser which requires an input technology library mapping file to recognize the primitive logic elements and requires an output technology library mapping file for technology transfer purpose.

2. It parses multiple files associated with the design, and each file may contain more than one module.

3. It supports top-down hierarchical design because it is designed with a hierarchical data structure to accommodate a hierarchical description of modules.

4. It converts the top-down hierarchical design to structural VHDL description without changing the hierarchical structure.

5. It flattens the hierarchical structure and generates the non-hierarchical TDN description consisting of the primitive logic elements.

6. It records the ignored components specified in *.ucomplist* system file. NDL Parser will treat them as primitives and print them out in TDN format.

In this section, we will first introduce the language NDL including its grammar and the data structure to accommodate the structure of the language, and then describe the parser.

```
# alu
# 15 inputs
# 2 outputs
# 3 D-type flipflops
# 18 gates (2 ANDs +4 NANDs +7 ORs +0 NORs + 3 XORs +2 XNORs + 0 BUFF )
 INPUT (acmp)
INPUT (tree)
INPUT (agetstreel)
INPUT (agetssuml)
INPUT (bcmp)
INPUT (bgetsmeml)
INPUT (bgetsenl)
INPUT (ccmp)
INPUT (cgetscarry)
INPUT (rddata)
INPUT (clock)
INPUT (ldcarry)
INPUT (ldenable)
INPUT (forceenable)
INPUT (memwrite)
OUTPUT (wrdata)
OUTPUT (writeenable)
u24_0(writeenable=z)=an02d1(forceenable=a1, l_1=a2)
u25_1##u0_0(l_54=z)=or02d1(l_3=a1, l_4=a2)
u25_1##u1_1(l_57=z)=or02d1(l_5=a1, l_6=a2)
u25_1##u2_2(l_7=zn)=nd02d1(l_54=a1, l_57=a2)
u26_2(l_5=z)=an02d1(l_4=a1, l_3=a2)
u27_3(l_4=zn)=xn02d1(ccmp=a1, l_12=a2)
u28_4(l_6=z)=xo02d1(acmp=a1, l_15=a2)
u29_5(l_3=z)=xo02d1(bcmp=a1, l_18=a2)
u30_6##u0_0(l_63=z)=xo02d1(l_3=a1, l_6=a2)
u30_6##u1_1(l_23=zn)=xn02d1(l_4=a1, l_63=a2)
u31_7(l_1=z)=or02d1(memwrite=a1, l_25=a2)
u32_8##u0_0(l_69=z)=or02d1(tree=a1, agetstreel=a2)
u32_8##u1_1(l_72=z)=or02d1(agetssuml=a1, wrdata=a2)
u32_8##u2_2(l_15=zn)=nd02d1(l_69=a1, l_72=a2)
u33_9##u0_0(l_78=z)=or02d1(rddata=a1, bgetsmeml=a2)
u33_9##u1_1(l_81=z)=or02d1(l_25=a1, bgetsenl=a2)
u33_9##u2_2(l_18=zn)=nd02d1(l_78=a1, l_81=a2)
u34_10(l_12=zn)=nd02d0(cgetscarry=a1, l_38=a2)
enable_reg_11(l_25=q, l_43=qn)=dfntnb(l_7=d, ldenable=cp)
wrdata_hold_reg_12(wrdata=q, l_47=qn)=dfntnb(l_23=d, clock=cp)
carryflag_reg_13(l_50=q, l_38=qn)=dfntnb(l_7=d, ldcarry=cp)
```

Figure 3.5: Testability Description Netlist (TDN)

```
compile ;
module module_name ;
inputs input_pin_name ;
outputs output_pin_name ;
define
        instantiation
        signal_declaration
end module;
        .
        .
        .
module module_name ;
inputs input_pin_name ;
outputs output_pin_name ;
define
        instantiation
        signal_declaration
end module;
end compile;
```

Figure 3.6: NDL skeleton files

3.4.1 A Brief Overview of NDL

NDL is often used as a description for the circuit netlist in ASIC designs. For instance, the *Testability Description Language (TDN)* used in the Mentor Graphics tools shares the same grammar as LSI Logic Inc. NDL. This gate-level logic design netlist is developed by LSI Logic and used in C-MDE environment to represent the gate-level netlist.

In the hierarchical top-down design, there must be one top-most module in which all components or submodules are used or instantiated. The interconnections among these submodules are explicitly defined within the body of the module. Then the submodules are defined individually for further details within the submodules. NDL is a language that supports this hierarchical design in terms of defining modules in each level. Modules consist of connected I/O pins and the primitive components. Also, modules may be user-defined modules and/or building blocks.

To define these modules, users must follow the syntax and grammar. The NDL netlist must have a skeleton like the one in Figure 3.6.

3.4.2 Data Structure Design

Since the NDL has the hierarchical feature to define a design, the NDL Parser must have a data structure that is established with similar features to efficiently and effectively capture the NDL description. Figure 3.7 shows the data structure for the NDL. A *component pool* is built using a linked list technique to hold all the components used in the design

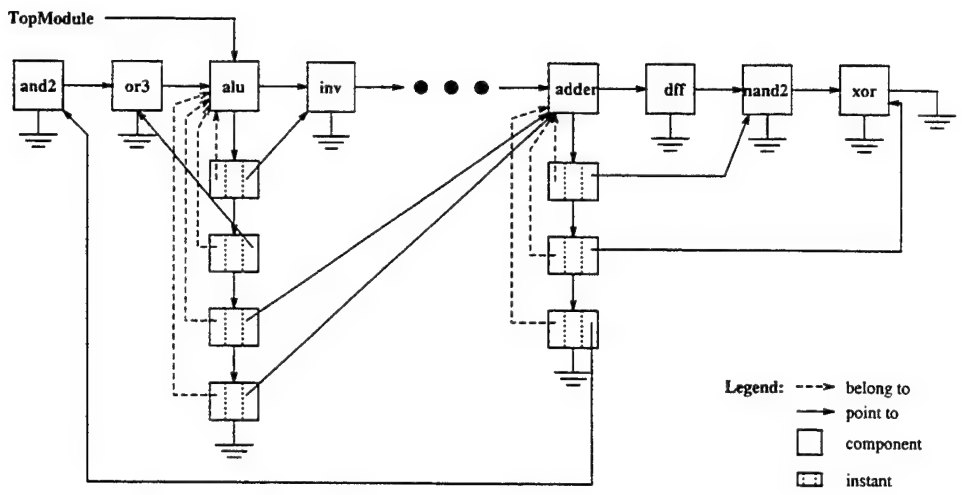


Figure 3.7: Data structure for the NDL Parser

regardless of the different hierarchical levels in which they reside. Each component carries its own component name, input and output port names called *local*, gate counts or cell units (or area information) of this component, the index of technology library look-up table (mapping file), the instance declared within the component. Within each component, all the sub-components are declared to be used through component instantiation. There is one top-most component in the *component pool*. It will carry the design name. The instance data structure contains two pointers; one is used to point to the immediate employing component that the current instant is instantiated, and the other is used to point to the component in the *component pool* tellng the nature of this instance. It also contains a list of port maps, that is, maps of the actual signal names under current component and port names on this instance.

3.4.3 System Flow of the NDL Parser

We use *Yet-Another-Compiler-Compiler* (YACC) to write the NDL Parser. The YACC allows Backus-Naur-Format (BNF)-like rules to be written to read in NDL description Since the NDL description is a multi-module description, all the NDL modules described in different files will be read in by the NDL Parser and the data structure described in the Section 3.4.2 will be established during the reading process. Figure 3.8 depicts the system flow of the NDL Parser.

3.5 VHDL Parser

The VHDL Parser also employs YACC to parse the design written in VHDL to an NDL description. Since VHDL is a large language, it is too difficult to develop a parser to cover all the constructs of VHDL. The VHDL Parser is then primarily developed to cover the structural description of VHDL. The BNF-like rules are written in YACC to cover the VHDL structural descriptions. The parser will capture the VHDL structural descriptions and establish the data structure as described in Section 3.5.1. The VHDL parser will primarily generate the NDL netlist. VTST employs the NDL Parser to flatten the design

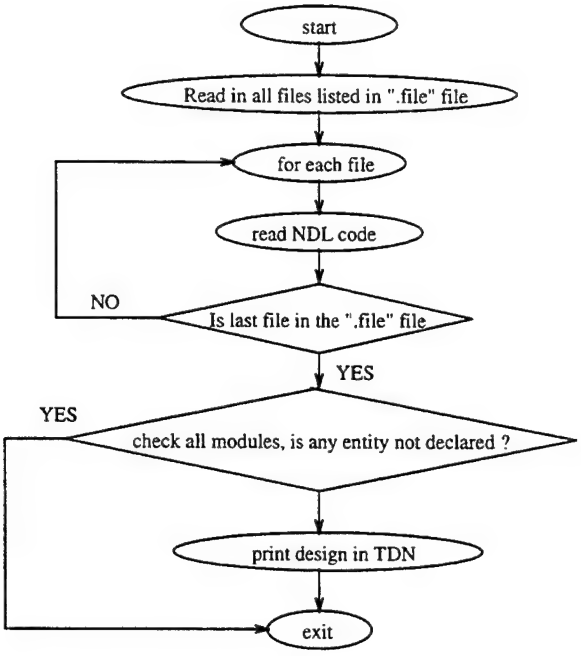


Figure 3.8: Flow chart for NDL parser

and generate the TDN format.

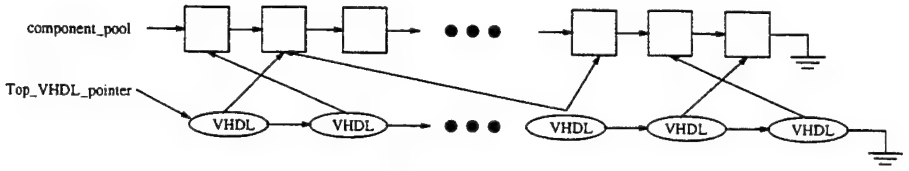
In this section, the data structure will be first introduced to capture the VHDL description. Then a system flow of the VHDL parser will be introduced.

3.5.1 Data Structure for VHDL Parser

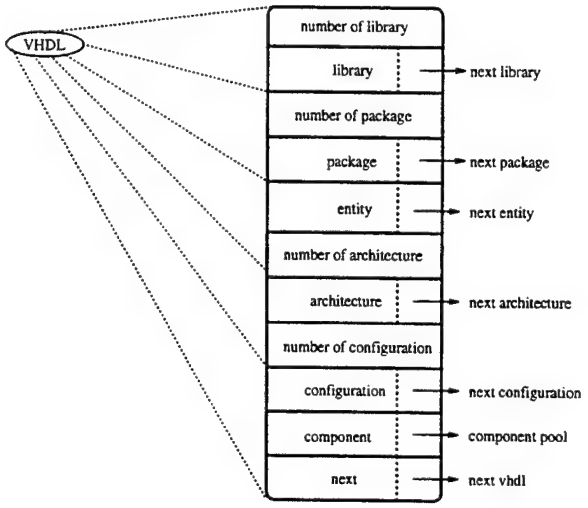
A VHDL component pool is established as all the files written in VHDL are read in. This pool will be a linked list of nodes. Each node is a component where the name of the component, the nature of the component, the function of the component, the gate count, the input and output ports of the components are described respectively. In Figure 3.9, the list of squares is the component list the VHDL Parser will build. Another linked list is also established for all VHDL entities. Each node in this linked list is an entity defined in VHDL files. Each entity node will point to a particular component in the component linked list. Furthermore, the content of each entity will include the selected library, the declared package, the architectures, the configurations, the components and finally the next VHDL entity node. Figure 3.9 shows the structure of each VHDL entity.

3.5.2 System Flow of the VHDL Parser

Figure 3.10 depicts the system flow of the VHDL Parser. The name of all the VHDL files related to the CUT will be first list in the system file ".file". VHDL Parser will read in all the files listed there using YACC based parser and establish the data structure. The parser will issue an error message to the user if there exists any unread entity. After finishing reading the file, the data structure will be established and the VHDL parser will generate



(a)



(b)

Figure 3.9: Data structure for VHDL parser

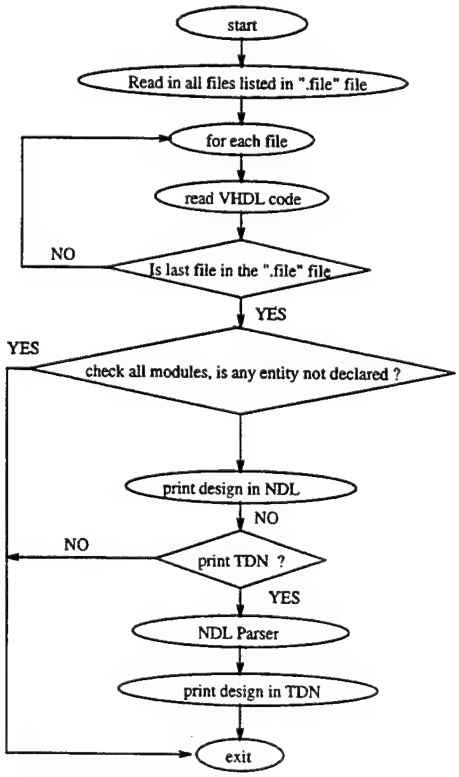


Figure 3.10: Flow chart of the VHDL parser

an NDL description of the design. Then VTST will employ the NDL Parser to read in the NDL netlist generated by the VHDL parser and generate the TDN format as requested for testability analysis.

Chapter 4

BIST Design and Testability Analysis Tools: (BISTDaTA)

The aim of developing VTST software tools is to increase the observability and the controllability of internal nodes so that the stuck-at faults present in the CUT can be detected. The Built-In Self-Test (BIST) Design and Testability Analysis (BISTDaTA) tool is a toolset consisting of different tools to perform the testability analysis and the generation of different BIST designs to meet the design requirements. Each tool developed serves a distinct purpose. These tools include:

1. A circuit preprocessor which is called *Senor/Scissor*,
2. A pseudo-scan processor which is called *Pseudo-Scan Scissor*,
3. A circuit Partitioning tool called *Autonomous*,
4. A test signal reduction synthesizer called BISTSYN,
5. A BIST tool for sequential circuits called *SEQBIST*,
6. A XOR-Tree generation program called XOR-Tree Gen,
7. A BIST design rebuild program called VTST BUILDER.

All tools developed in BISTDaTA require a circuit input written in the TDN format as described in a previous chapter and will produce several output files. Each file produced also serves a significant purpose for further testability analysis, BIST hardware insertion, and tool output information. These files are:

1. A circuit output file written in TDN (NDL for VTST BUILDER).
2. A file of actions which is required for the insertion of BIST hardware.
3. A tool report which presents the results of BISTDaTA.
4. A tool output file which contains tool output information.

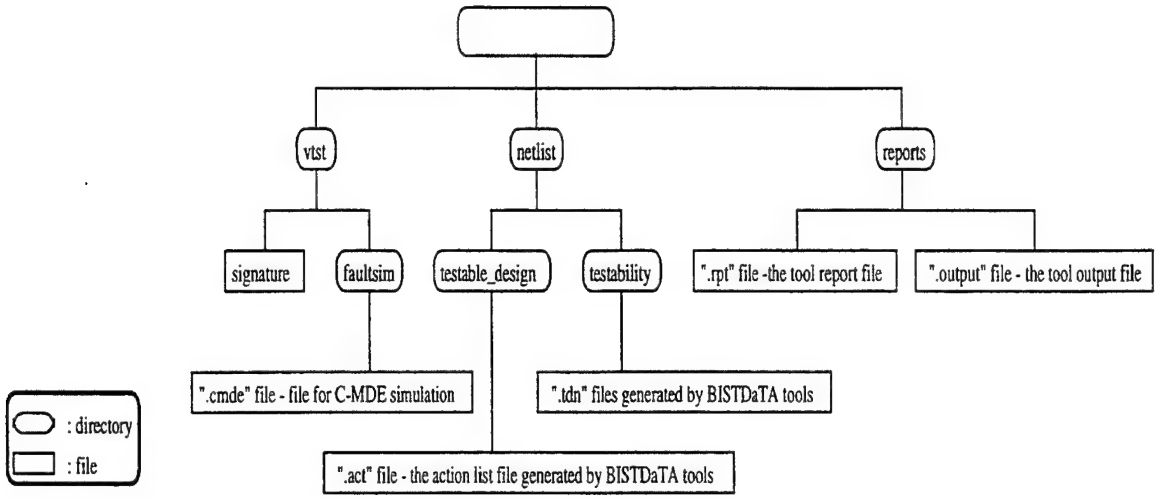


Figure 4.1: File hierarchy of BISTDaTA tools

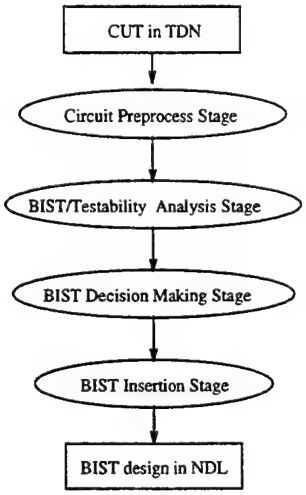


Figure 4.2: BISTDaTA system flow chart

4.1 File Hierarchy of BISTDaTA tools

Work directory defaults to be the directory where the VTST tool is invoked. In the work directory, there are three sub-directories that will be created by the VTST tool, the *vtst* directory, the *netlist* directory, and the *reports* directory. Figure 4.1 outlines the vtst file hierarchy. In the *vtst* directory, the tool output files will be generated where a subdirectory called *Faultsim* will be generated to record all the simulation files generated by VTST for C-MDE simulation use. In the *netlist* directory, there are two key subdirectories: the *testable_design* and the *testability* subdirectories. In the *testable_design* subdirectory, all the action lists generated by BISTDaTA will be written and the VTST rebuild program will read in these action lists for generating a final action list for the BIST design rebuild process. In the *testability* subdirectory, all the output circuit files generated by BISTDaTA tools will be written. All the BISTDaTA tools will read in these files for further testability analysis.

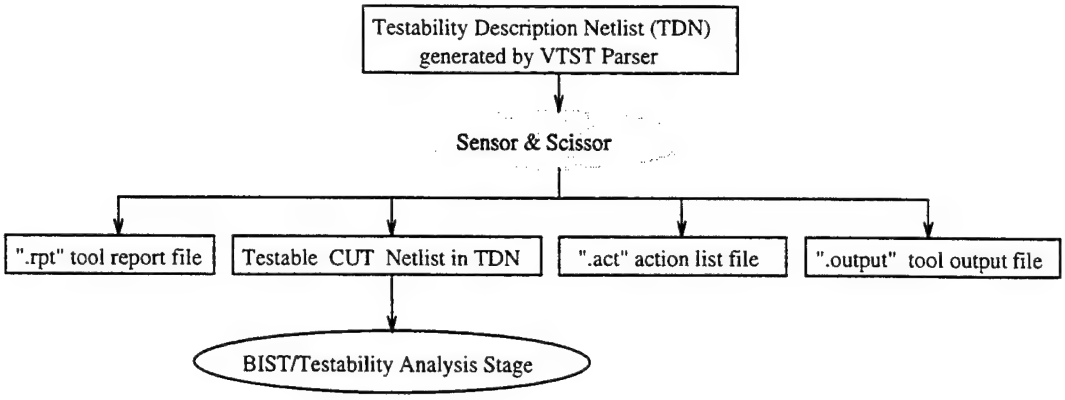


Figure 4.3: Circuit preprocess stage

4.2 System Flow

To make better usage of the BISTDaTA toolset, understanding the system flow of the BISTDaTA and the distinct features and purpose of each tool are necessary. Figure 4.2 depicts the systematic flow of BISTDaTA. There are four basic stages in BISTDaTA. After the parsing stage achieved by the VTST Parser, the CUT written in TDN format will be preprocessed by circuit preprocessor - *Sensor/Scissor* in the preprocessing stage. The CUT will be regenerated in the TDN format with all the testable features added to it. The testable CUT then is fed to the BIST/testability analysis stage for fault coverage calculation using different BIST design tools. BIST decision-making is a stage in which VTST users will determine which BIST design tool will be applied to the CUT so that the fault coverage and test length will meet or be close to the specification. Finally, a BIST design of the CUT will be generated in the BIST insertion stage based on the decisions made in BIST decision-making stage. The BIST design of the CUT will be produced in NDL netlist for the later stage of simulation and verification purposes.

4.3 Circuit Preprocessing Stage

In this stage, a CUT generated in TDN by the VTST Parser will be analyzed to re-structure or transform the tristate buffers, tristate buses, gated clocks, set/reset flip-flops and latches, and bidirectional input/output ports to testable design structures. All the flip-flops without set/reset features will be replaced with flip-flops with set/reset features. An action list will be generated to provide BIST design insertion actions to be used by the VTST rebuild program. A tool output file and tool report file will be generated to report the results of tool applied.

4.4 BIST/Testability Analysis Stage

BIST/Testability analysis stage is the most critical stage in the BISTDaTA analysis. In this stage, the testable CUT described in TDN will undergo different and distinct paths of

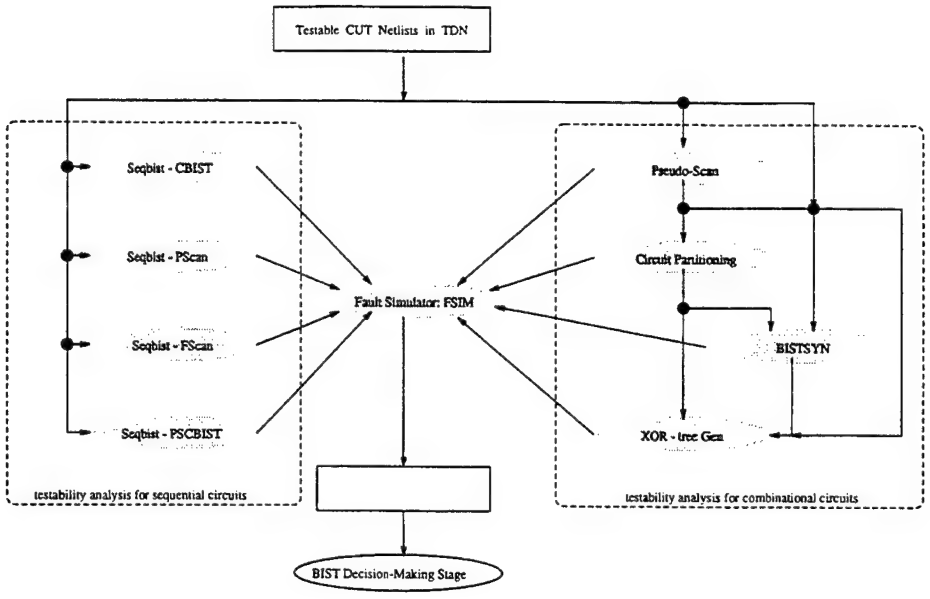


Figure 4.4: BIST analysis stage

testability analysis depending upon what BISTDaTA tool is selected for application to the CUT. Figure 4.4 shows an overview of the BIST/Testability analysis stage where several BISTDaTA tools may be involved in performing testability analysis. These tools include: Pseudo-Scan Scissor, BISTSYN, XOR-tree Generator, SEQBIST, and the VTST fault simulator FSIM.

In general, a circuit can be classified broadly into combinational circuits or sequential circuits. A combinational circuit refers to random logic without memory elements such as flip-flops/latches. On the other hand, a sequential circuit is a finite state machine or a random logic with memory elements. Testing a sequential circuit is very different from testing a combinational circuit. Simply because the primary outputs of sequential circuits depend upon not only the primary inputs but also upon the previous states of the memory elements.

In the BIST/Testability analysis stage, there are two paths for testability analysis. One is the path for sequential circuits while the other one is for combinational circuits. However, it doesn't mean that a testable sequential circuit after *Sensor/Scissor* must undergo the path for sequential circuits. *Pseudo-Scan Scissor* tool in the BISTDaTA transforms a sequential circuit to a combinational circuit virtually during testing. After the transformation, test patterns generated by the SG module can be directly applied to the CUT. For a detailed description of the *Pseudo-Scan Scissor* tool, refer to Chapter 6.

4.4.1 Testability Analysis for Combinational Circuits

For combinational circuits, there are two paths for testability analysis. They are:

1. Applying BISTSYN to the testable combinational circuit generated by the VTST Parser to reduce the test signals required for pseudo-exhaustive testing; and then applying the XOR-tree Gen to the CUT for the reduction of size of MISR for signature

analysis.

2. Applying AUTONOMOUS to the testable combinational circuit generated by the VTST Parser to reduce the primary output dependency of the CUT such that the size of the test signals required will be further reduced; and then applying the XOR-tree Gen to the CUT for the reduction of the size of the MISR for signature analysis.

4.4.2 Testability Analysis for Sequential Circuits

For combinational circuits, there are several paths for testability analysis. They are:

- Transforming the sequential circuit to pseudo combinational circuits by removing all the flip-flops and latches (register-free option) so that testability analysis for combinational circuits can be applied. Then, apply BISTSYN to the testable combinational circuit generated by the VTST Parser to reduce the test signal required for pseudo-exhaustive testing; and then applying the XOR-tree Gen to the CUT for the reduction of size of the MISR for signature analysis; or applying AUTONOMOUS to the testable combinational circuit generated by the VTST Parser to reduce the primary output dependency of the CUT so that the size of test signals required will be further reduced, and then apply the XOR-tree Gen to the CUT for the reduction of size of MISR for signature analysis.

- Transforming the sequential circuit to pseudo combinational circuits by removing all the flip-flops in the CUT and all the latches in feedback paths (flipflop-free option) so that testability analysis can be applied. Then, apply BISTSYN to the testable combinational circuit generated by the VTST Parser to reduce the test signal required for pseudo-exhaustive testing; and then applying the XOR-tree Gen to the CUT for the reduction of size of the MISR for signature analysis; or applying AUTONOMOUS to the testable combinational circuit generated by the VTST Parser to reduce the primary output dependency of the CUT so that the size of the test signals required will be further reduced, and then apply the XOR-tree Gen to the CUT for the reduction of size of MISR for signature analysis.

- Transforming the sequential circuit to pseudo combinational circuits by removing all the flip-flops and latches in feedback paths (feedback-free option) so that testability analysis can be applied. Then, apply BISTSYN to the testable combinational circuit generated by VTST Parser to reduce the test signal required for pseudo-exhaustive testing; and then applying the XOR-tree Gen to the CUT for the reduction of size of the MISR for signature analysis; or applying AUTONOMOUS to the testable combinational circuit generated by the VTST Parser to reduce the primary output dependency of the CUT so that the size of the test signals required will be further reduced, and then apply the XOR-tree Gen to the CUT for the reduction of size of the MISR for signature analysis.

- Apply SEQBIST with the circular BIST (CBIST) option and then apply the XOR-tree Gen to the CUT for the reduction of the size of the MISR for signature analysis.

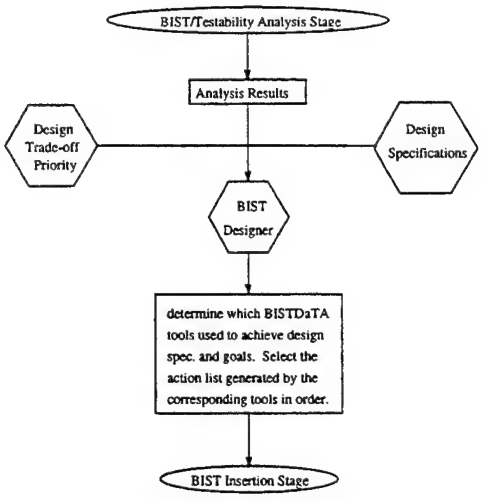


Figure 4.5: BIST decision making stage

- Apply SEQBIST with the full scan (FSCAN) option and then apply the XOR-tree Gen to the CUT for the reduction of the size of the MISR for signature analysis.
- Apply SEQBIST with the partial scan (pscan) option and then apply the XOR-tree Gen to the CUT for the reduction of the size of the MISR for signature analysis.
- Applying SEQBIST with CBIST with the pseudo-partial scan (CBISTPPSCAN) option and then apply the XOR-tree Gen to the CUT for the reduction of the size of the MISR for signature analysis.

4.4.3 Fault Coverage Estimation

All the paths described in Subsection 4.4.1 and 4.4.2 provide different means for VTST users to perform different BIST design methodologies to improve the controllability and observability of the CUT. To evaluate the fault coverage, a fault simulator is employed.

4.5 BIST Decision Making Stage

In BIST/Testability analysis stage, analysis results and action list files will be generated after the BIST tool is applied. The results provided will be used by the VTST users to determine which analysis path done in the BIST/Testability analysis stage will achieve the desired design specifications and reasonable design trade-offs. Users can always return to the BIST/Testability analysis stage to look for a workable analysis path to improve the design. After the users have determined the BIST design methodologies, the action lists associated with the design tools will be selected by the VTST action generator according to the order of the tools applied to the CUT to generate a final action list file for the VTST rebuild program use. BUILDER will then generate the BIST design of the CUT. Figure 4.5 depicts the BIST Decision-Making Stage.

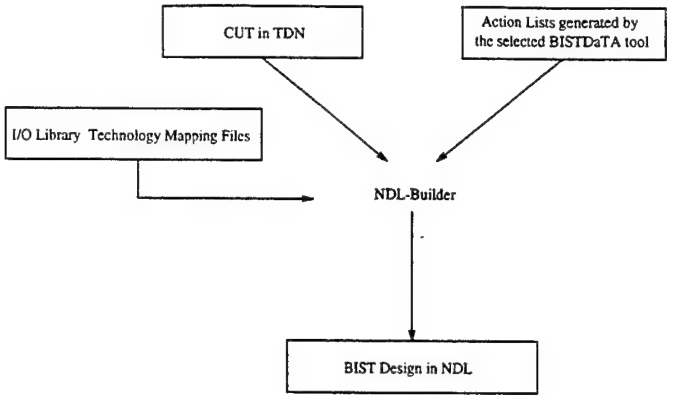


Figure 4.6: BIST insertion stage

4.6 BIST Insertion Stage

In the previous stages, the BIST design methodologies have been determined and applied. In this stage, the BIST test hardware will be inserted according to the instructions listed in the final action list file. Furthermore, the SG module, RA&C module, FSG module, all the system Mux's and all the interconnections among these modules and CUTs will be added to the CUT to complete the BIST insertion stage. Finally, the BIST design of the CUT will be generated in NDL netlist format. Through the NDL Parser, the BIST design of the CUT can also be regenerated in VHDL. Figure 4.6 depicts the BIST Insertion Stage flow chart.

4.7 Summary

The BISTDaTA program is developed in C. The flow charts shown in Figures 4.7 and 4.8 depict the detailed flow of BISTDaTA tool.

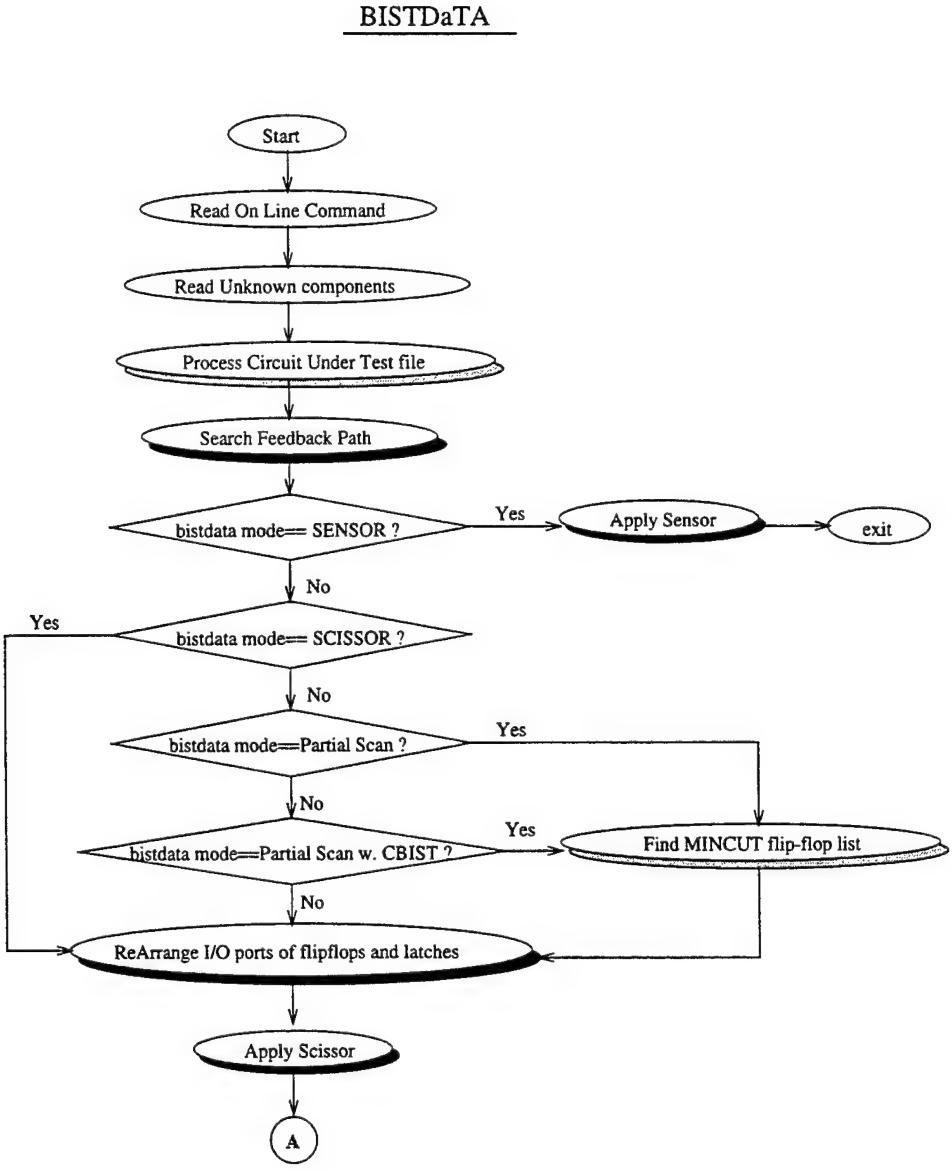


Figure 4.7: BISTDaTA flow chart 1

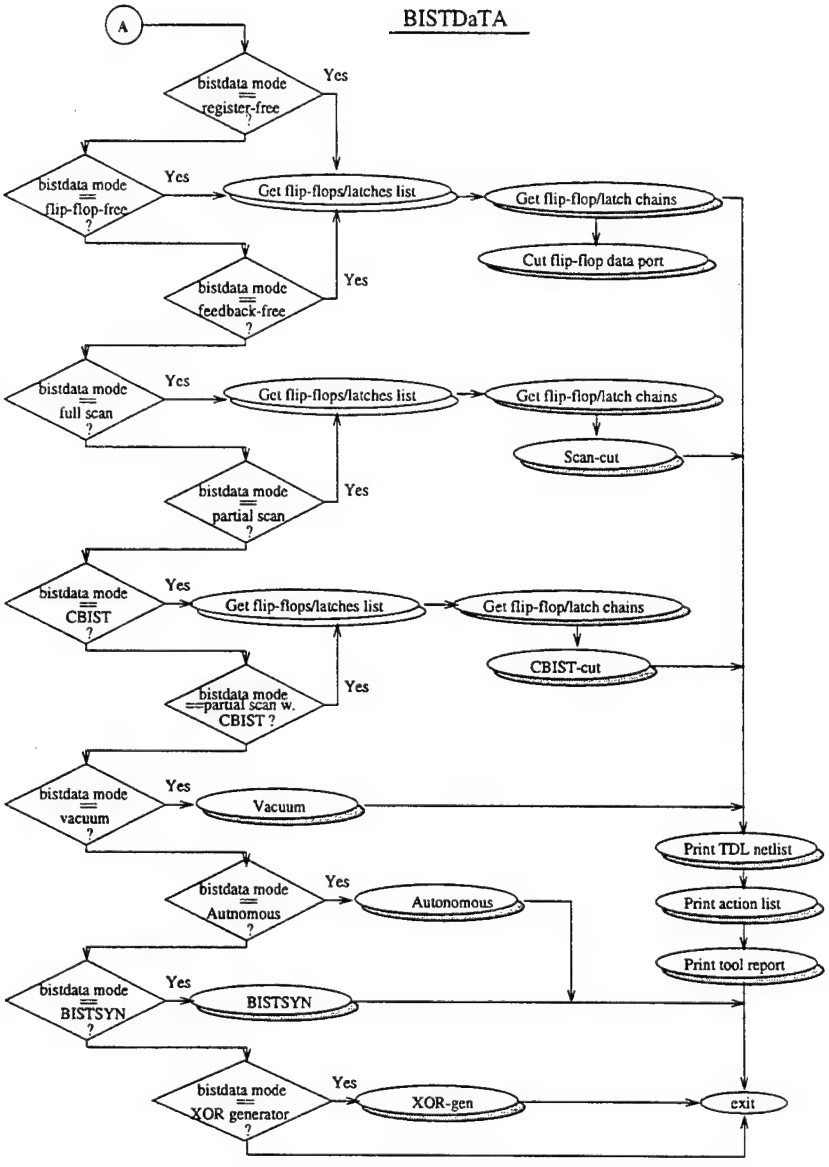


Figure 4.8: BISTDaTA flow chart 2

Chapter 5

Circuit Analysis and Preprocessing Tool: Sensor-Scissor

A testable design is a design in which all or almost all the stuck-at faults are testable. Designing a testable circuit is very desirable today and in future VLSI circuits as their structures are getting more and more complex. Tristate buffers, tristate buses, and gated clock flip-flops/latches are those designs that often make the CUT untestable. To deal with these designs without altering the normal operation, a tool called *Sensor-Scissor* has been developed and included in BISTDaTA to process the CUT for testability analysis. It is a software program which will read in the CUT described in TDN, perform testability restructure, and regenerate the CUT in TDN.

5.1 A Tool for Circuit Analysis - Sensor

Sensor-Scissor provides the capability of detecting the special circuitry such as tristate buffers, different types of tristate buses, bidirectional inputs/outputs, flip-flops and latches that are residing in the feedback path. The sensor tool analyzes the CUT and gives the VTST user a preview of the nature of the CUT. Figure 5.1 is a report generated by the Sensor tool, which will appear on the screen when the tool is invoked.

5.2 A Tool for Circuit Preprocess - Scissor

The tool *Scissor* is developed to restructure or modify the circuit that consists of the following gates so that the CUT becomes testable.

- tristate buffers,
- tristate buffer buses with the following four different cases:
  - the enables of all tristate buffers in a tristate bus are from the same source, and the inputs of all these tristate buffers are also from the same source.
  - the enables of all tristate buffers in a tristate bus are from the same source but the inputs of these tristate buffers are from different sources.

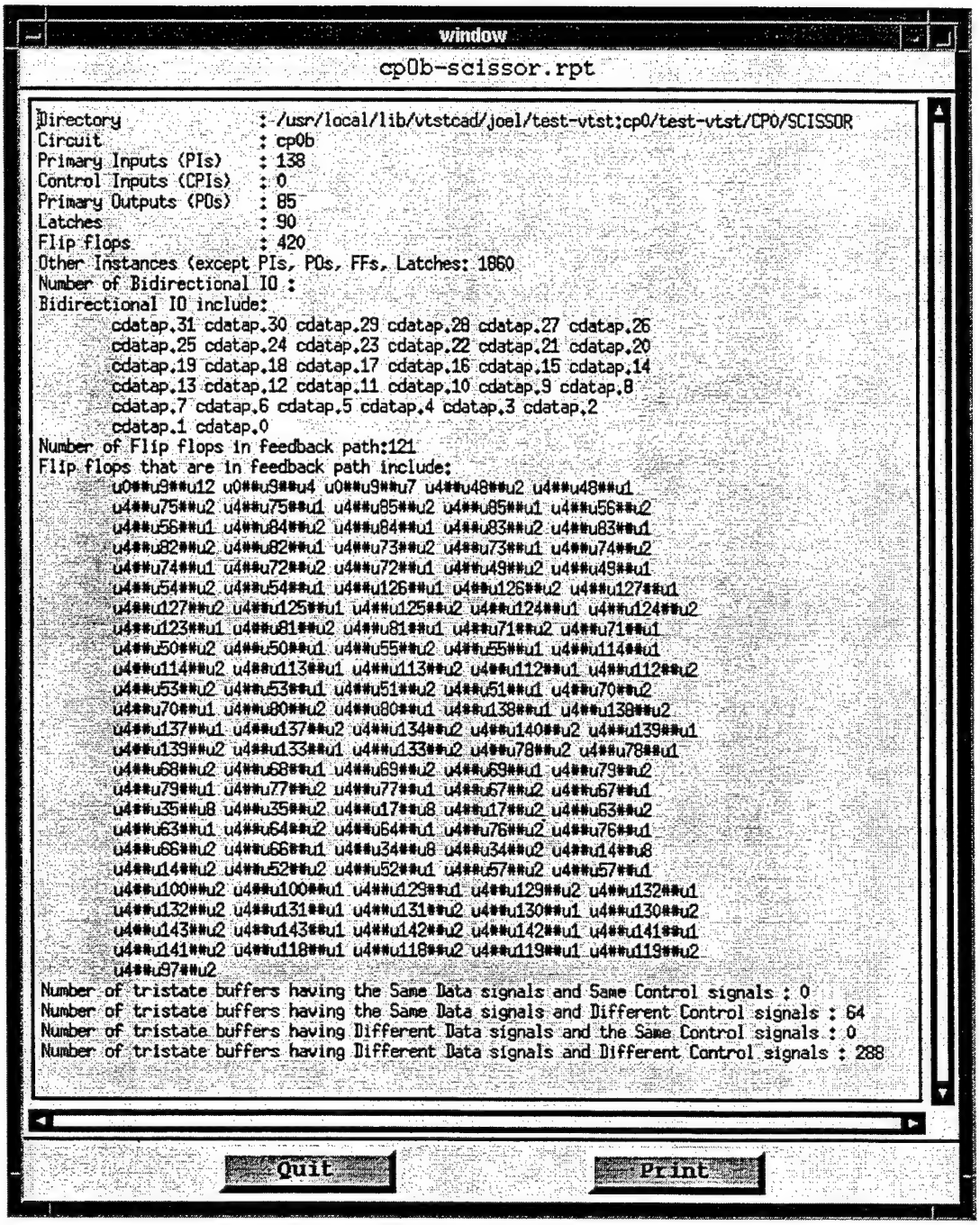


Figure 5.1: A window report generated by Sensor-Scissor program

- the inputs of all tristate buffers in a tristate bus are from the same source but the enables of these tristate buffers are from different sources.
- both the enables and the inputs of all the tristate buffers in the tristate bus are from different sources.

- gated clock, set, reset flip-flops and latches.
- bidirectional input/output ports.

The methodologies used for testing tristate buffers, tristate buses, and bi-directional inputs/outputs are absolutely different although they all consist of tristate buffers.

5.2.1 Testing Tristate Buffers

A tristate buffer serves like a regular non-inverting buffer when it is enabled. However, the output of the tristate buffer will be high impedance when it is disabled. Figure 5.2(a) and Figure 5.2(c) depict a LOW activated and HIGH activated tristate buffer respectively. It is not desirable to have high impedance at the output during testing. Therefore, it is desirable to enable a tristate buffer and make it to be transparent during testing. To do that a Mux which is controlled by the signal *vtst_mux_control* is inserted before the enable signal of the tristate buffer. During testing, it is treated as a regular buffer for fault analysis. On the other hand, it can also serve as a regular tristate buffer in normal operation. Figure 5.2(b) and Figure 5.2(d) depict the testable structure of a LOW activated and HIGH activated tristate buffers respectively.

5.2.2 Testing Tristate Bus

The tristate bus is a group of tristate buffers whose outputs are wired together to a single output. However, the input of each tristate buffer and the enable signal for each one may not necessarily come from the same source. Figures 5.3(a), 5.4(a), 5.5(a), and 5.6(a) depict the four different cases of tristate buses. Unlike testing a tristate buffer, testing a tristate bus is implemented in different ways. During testing, all the tristate buffers in the tristate bus are isolated from the circuit. The enable signals controlling all the tristate buffers are pulled out as an observation point and a Mux is inserted to apply a HIGH or LOW signal, depending upon whether the tristate buffer is HIGH or LOW activated, in order to disable the tristate buffer. Then the inputs to the input data ports of all the tristate buffers are pulled out as primary output to be observed while a Mux is inserted between the output of the tristate bus and the next gate which the bus fanouts to. There are four distinct tristate buses. The test methodologies are very similar but with minor differences.

1. **Tristate bus with data ports from the same source and enable ports from the same source:** all tristate buffers are disabled during testing, a primary output (PO) is added to observe the data input to the tristate buffer and another primary output is added to observe the source which controls the tristate buffer. On the other hand, a primary input is inserted via a Mux to apply a test signal to the fanout(s) of the tristate bus (next logic). Figure 5.3 depicts the testable structure.

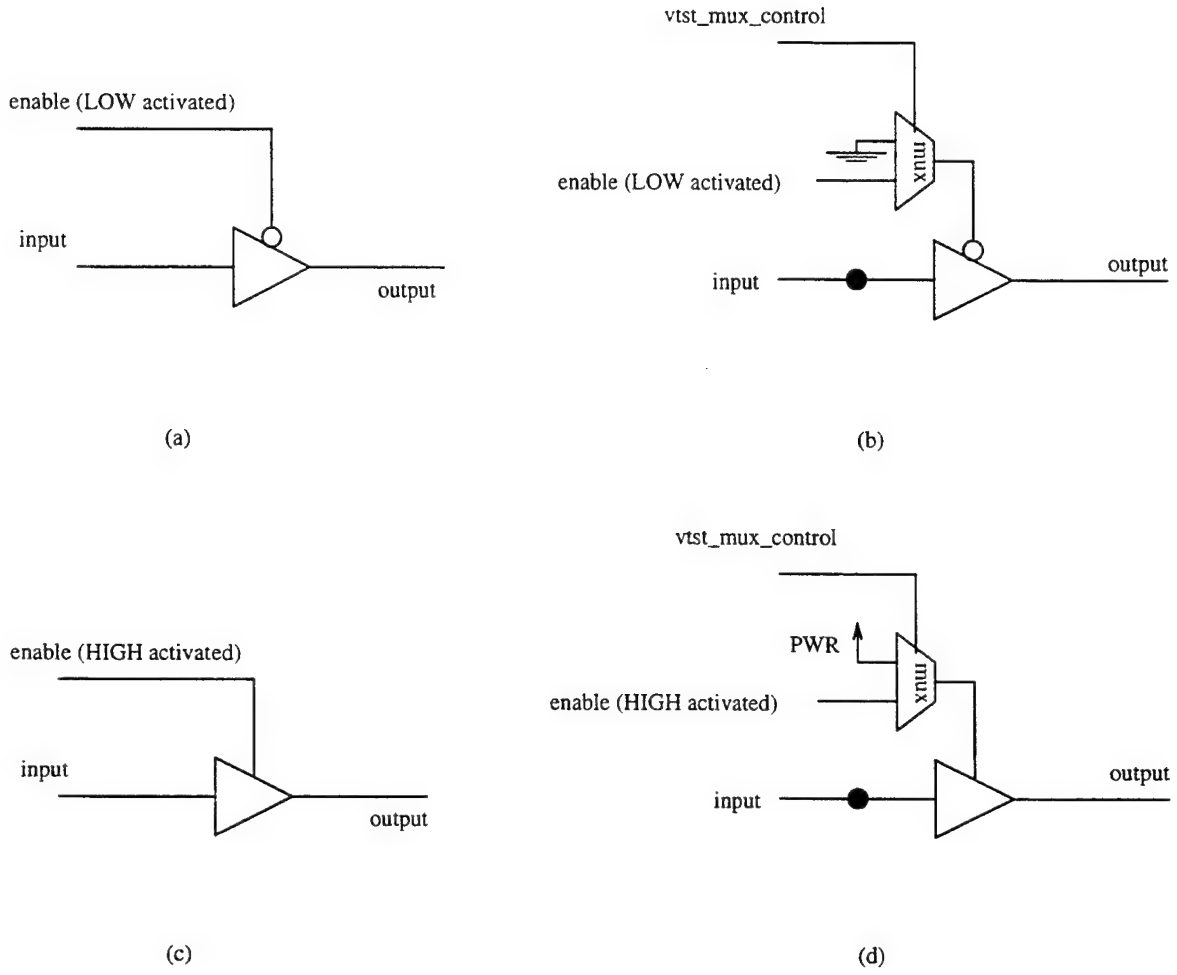


Figure 5.2: Testable design for a tristate buffer

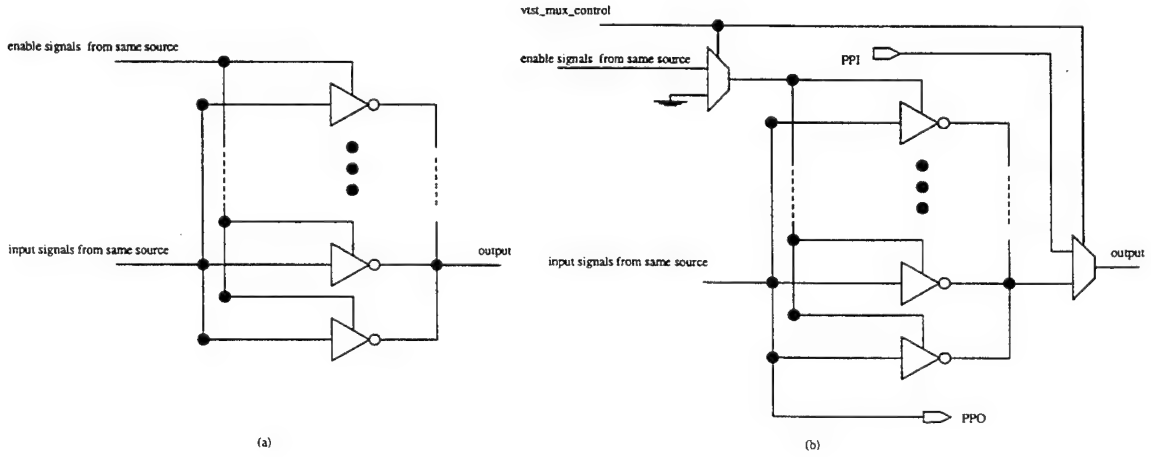


Figure 5.3: Tristate bus with inputs and enables from the same source

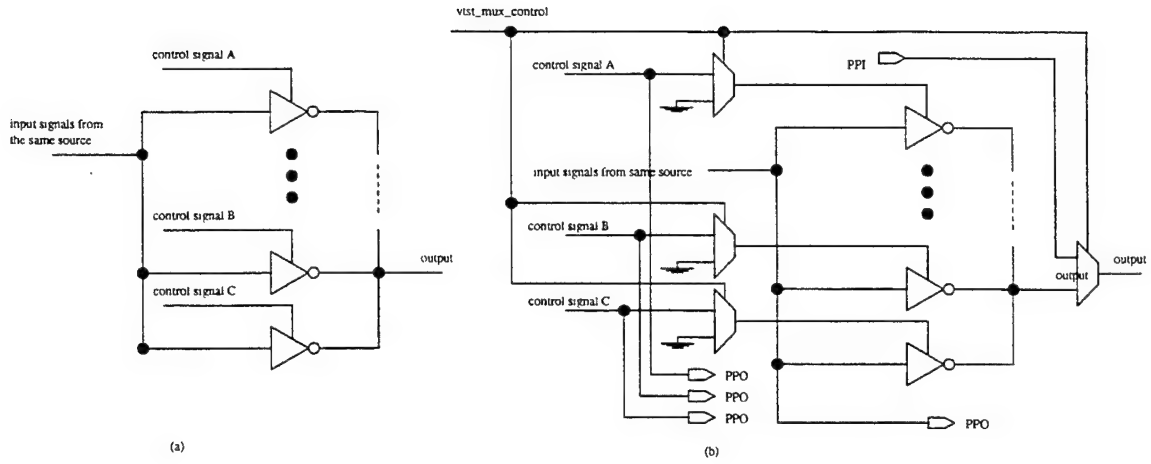


Figure 5.4: Tristate bus with inputs from the same source but enables from different source

2. **Tristate bus with data ports from the same source but the enable ports from different sources:** each source controlling the enable port of all the tristate buffers will be pulled out as one of the primary outputs to be observed. Mux's are inserted to apply HIGH or LOW signals to disable all tristate buffers during testing. A primary output is added to observe the input data port to the tristate buffers. A primary input is inserted via a Mux to apply a test signal to the fanout(s) of the tristate bus (next logic). The testable structure is shown in Figure 5.4.

3. **Tristate bus with data ports from different sources but the enable ports from the same source:** the source controlling the enable port of all the tristate buffers will be pulled out as a primary output to be observed. A Mux is inserted to apply a HIGH or LOW signal to disable all tristate buffers during testing. Primary outputs are added to observe all input data ports to the tristate buffers. A primary input is inserted via a Mux to apply a test signal to the fanout(s) of the tristate bus (next logic). The testable structure is shown in Figure 5.5.

4. **Tristate bus with data ports from different sources and the enable ports from different sources:** the source controlling the enable ports of all the tristate buffers will be pulled out as primary outputs to be observed. Mux's are inserted to apply HIGH or LOW signals to disable all tristate buffers during testing. Primary outputs are added to observe all input data ports to the tristate buffers. A primary input is inserted via a Mux to apply a test signal to the fanout(s) of the tristate bus (next logic). The testable structure is shown in Figure 5.6.

5.2.3 Bidirectional Input/Output Ports

Bi-directional input/output ports as shown in Figure 5.7(a) in the VTST test environment are treated as primary inputs. During testing, the output mode of bidirectional port is disabled and only the input mode is available for applying test patterns via the bidirectional port. However, this is not just adding a MUX to the bidirectional I/O because by doing that, the nature of the bidirectional I/O will be altered (a bidirectional I/O port will

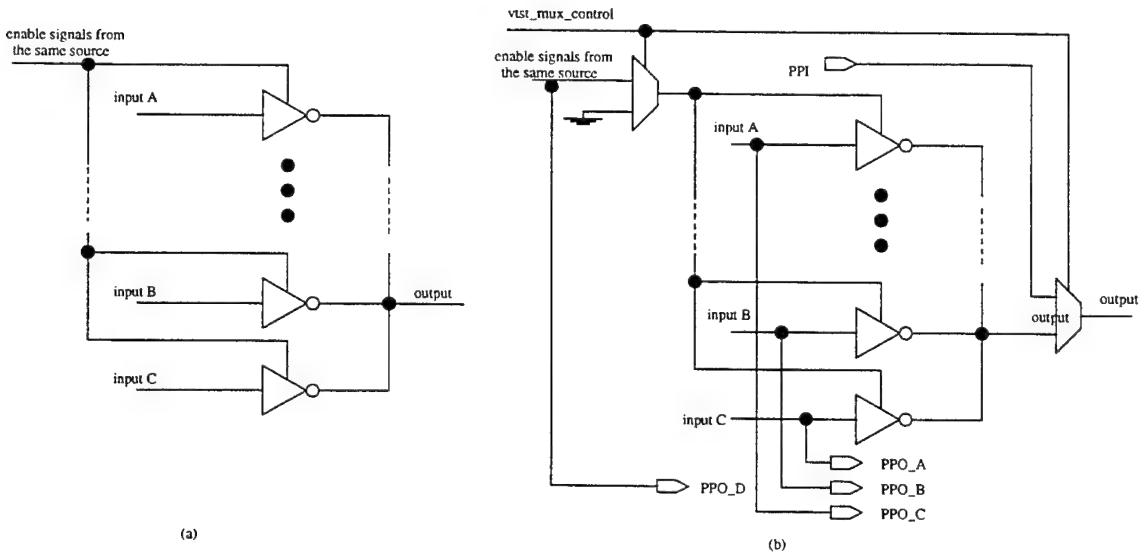


Figure 5.5: Tristate bus with inputs from different source but enables from the same source

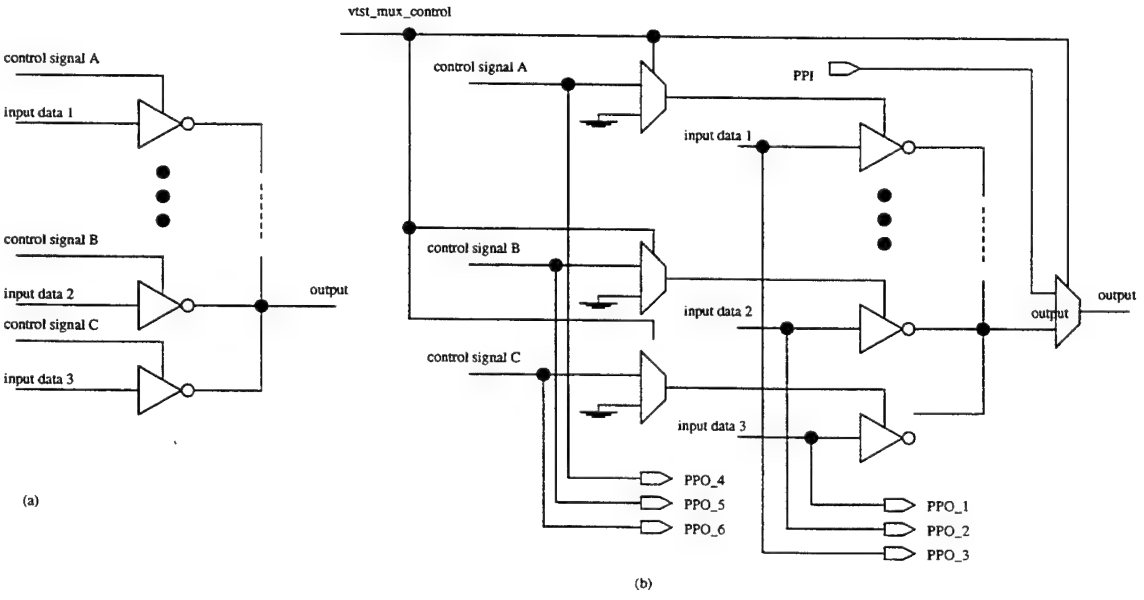


Figure 5.6: Tristate bus with inputs and enables from different source

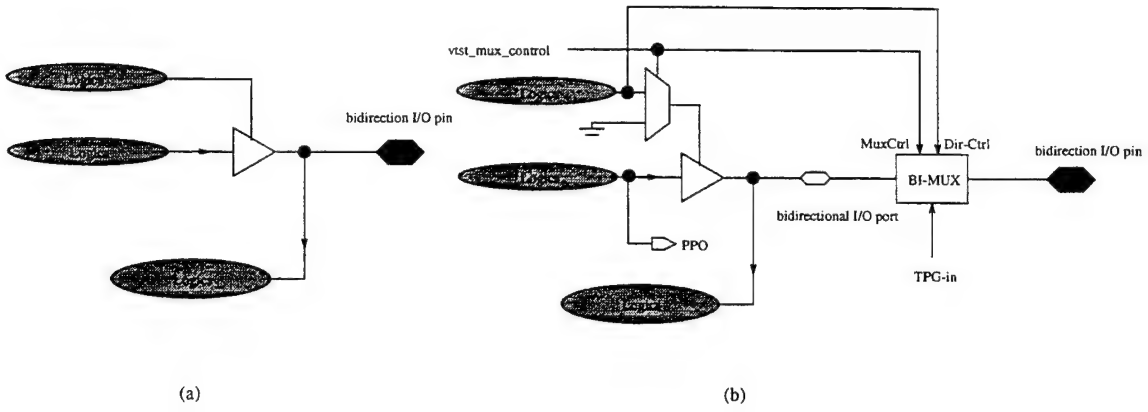


Figure 5.7: Bidirectional Input/Output Port

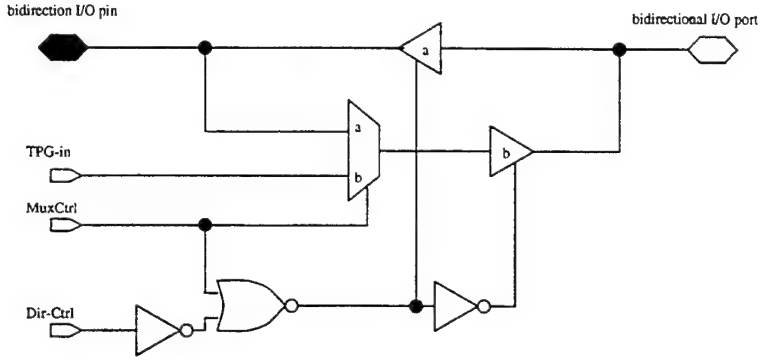


Figure 5.8: Bidirectional Multiplexers

become an input port). A special test hardware called *bidirectional MUX* is designed and developed to retain the bidirectional features while the output mode of the bidirectional I/O is disabled during the test mode. Figure 5.7(b) shows how a *bidirectional MUX* which is depicted in Figure 5.8 is inserted between the bidirectional I/O port and bidirectional I/O pin to retain its bidirectional nature. As shown in Figure 5.8, there are three input ports and two bidirectional I/O ports. The three input ports are *Dir-Ctrl*, *vtst_mux_control*, and *TPG_in*. One bidirectional I/O port will go to the bidirectional I/O pin while the other one will go to the circuit. Since a bidirectional port allows data to flow in and flow out via the I/O pin by enabling or disabling the tristate buffer associated with the bidirectional I/O port, the enabling/disabling signal for tristate buffer will be also employed to regulate the tristate buffers *a* and *b* used in the *bidirectional MUX* during the testing (that is why a NOR gate is used to drive both the signals *vtst_mux_control* and *Dir-Ctrl* to ensure *Dir-Ctrl* signal only takes effect during the test mode). The signal *vtst_mux_control* will direct the signal flow from the bidirectional I/O pin to bidirectional I/O port via a MUX and tristate buffer *b* or to the bidirectional I/O pin from bidirectional I/O port via state buffer *a* during the normal operation. For test mode, Signal from *TPG_in* will be selected and applied to the CUT via bidirectional I/O port.

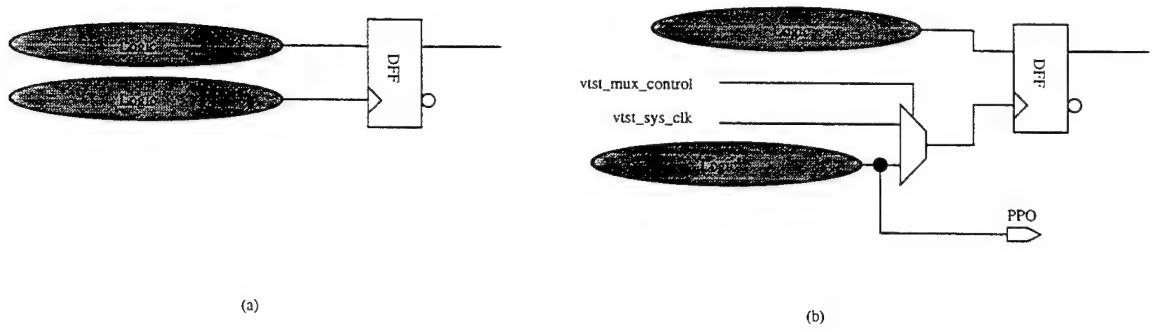


Figure 5.9: Gated clock D-type flip-flop

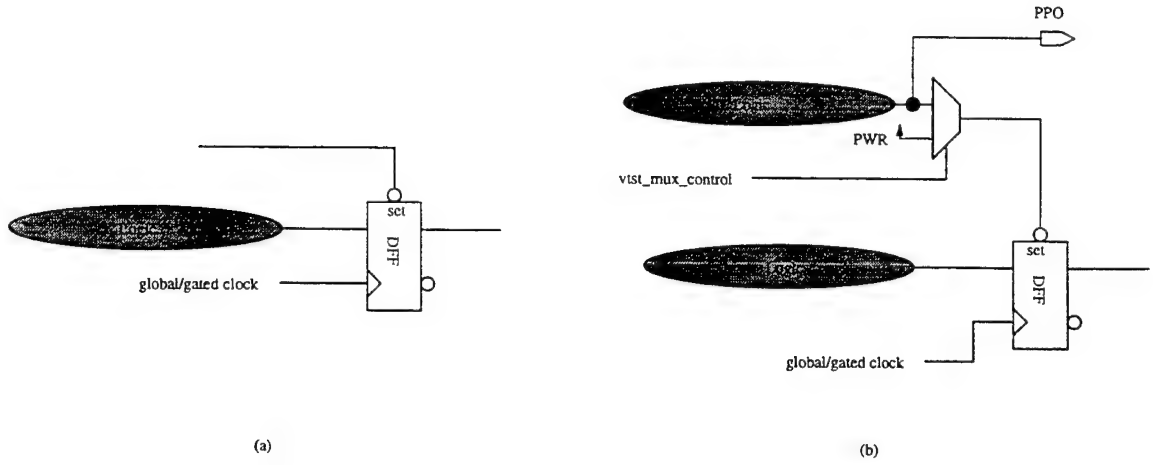


Figure 5.10: Gated clock D-type flip-flop

5.2.4 Gated Clock, Gated Set, Gated Reset Flipflops and Latches

Gated clock flip-flops and latches are special features of asynchronous design. Thery are not desirable in design for test. In test environment, all the flip-flops and latches are desired to be controlled by global control signals. Therefore, Mux's are added to the gated clock, set, and reset port to allow these elements to be accessed by global control signals during testing while their normal operation is resumed in normal mode. Also all the control signals which are generated by the circuit to control the gated clock, set, and reset of flip-flops and latches are pulled out as observation points. These are done by the *Sensor-Scissor* program and the testable structures generated by *Sensor-Scissor* program can be described schematically in Figures 5.9, 5.10, 5.11, 5.12, 5.13 and 5.14. For latches, they are made transparent during testing.

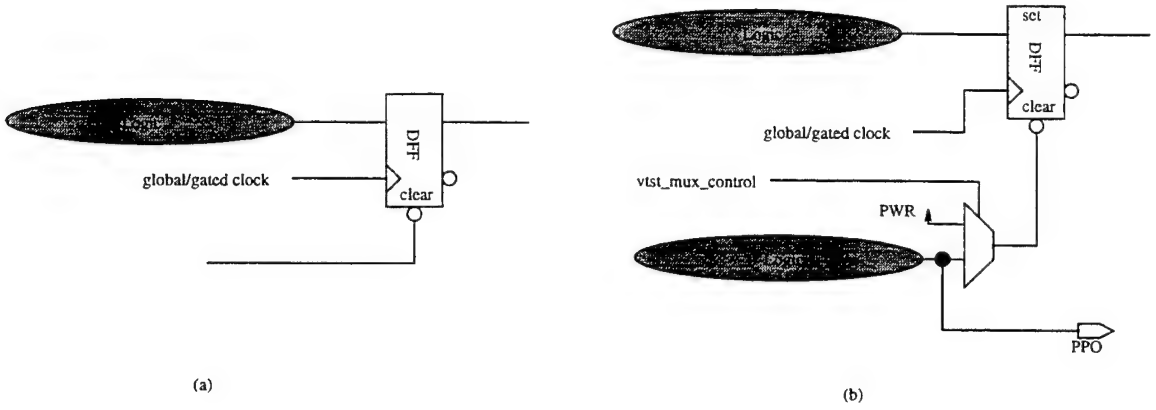


Figure 5.11: Gated clock D-type flip-flop

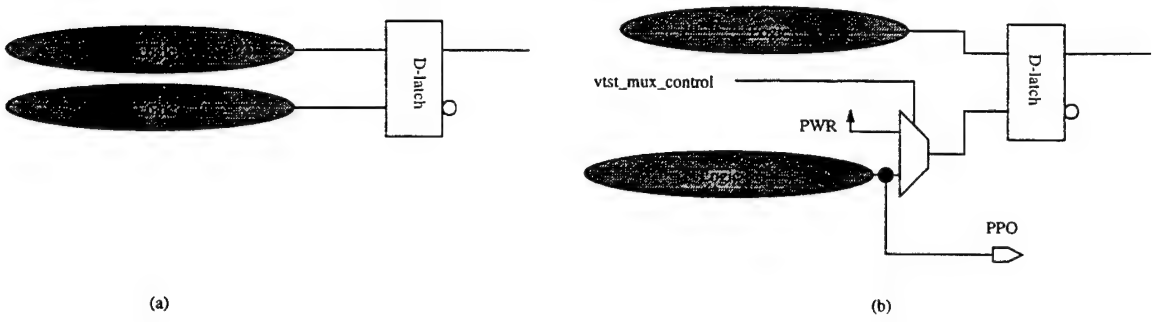


Figure 5.12: Gated enable D-type latch

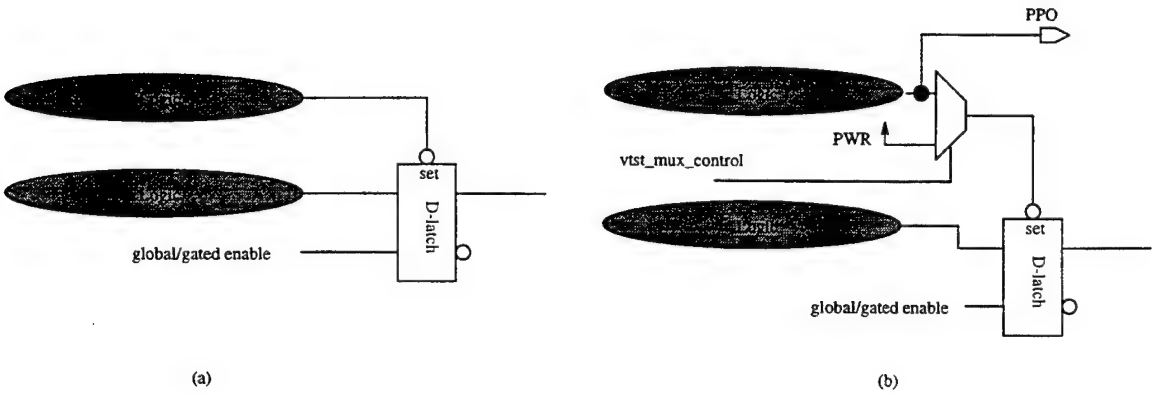


Figure 5.13: Gated set D-type latch

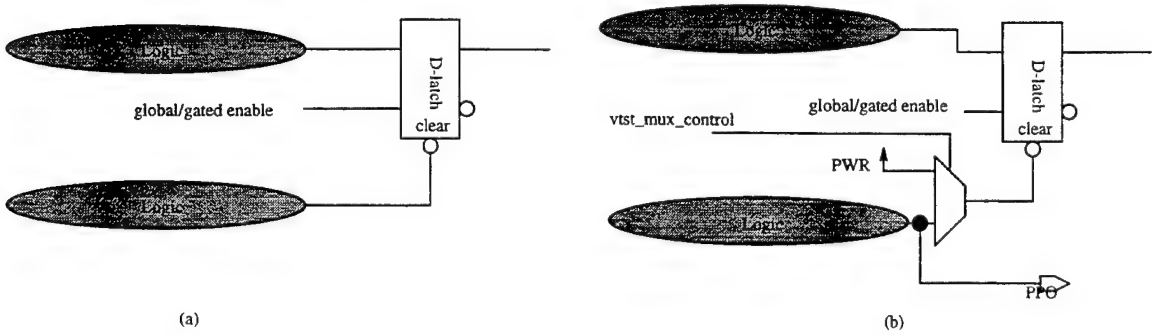


Figure 5.14: Gated clear D-type latch

Chapter 6

Pseudo-Scan Processing Tool: Pseudo-Scan Scissor

In general, circuits are primarily classified into combinational and sequential circuits. Any circuit which contains memory elements such as flipflops are considered as sequential circuits which are also named finite state machines since the primary outputs depend upon not only the primary inputs but also the previous states of the memory elements. Figure 6.1 describes a typical sequential circuit in which the memory elements are not necessary residing in the feedback path as shown in the figure. The circuit is reorganized so that the random logic are isolated from the memory elements for testability analysis purpose.

Testing a sequential circuit is very different from testing a combinational circuit. Testing a combinational circuit can be described as follows. In a particular time frame, a test pattern is applied and the response of the CUT is collected. A series of test patterns are applied sequentially (a series of time frames), but the responses in each time frame are exclusively determined by the test patterns applied to the CUT. The responses can be compressed and a signature can be generated. On the other hand, testing sequential circuits depends upon not only the test patterns applied to the CUT but also upon the previous states of the circuit. In a particular time frame, a test pattern is applied to the primary input of the CUT, the CUT will generate the responses. Part of the responses will become part of the inputs of certain portion of the logic in the next time frame. As illustrated in the Figure 6.1, flipflops can be in a chain or nested chain. The primary outputs are not only dependent upon previous states but also the previous state of the previous state.

Gaining access to the internal flipflops and latches becomes the essence of testing sequential circuits. The notion of gaining access is to have the test patterns loaded in the internal flipflops, apply them to the CUT, and then collect the responses from the CUT. For the chained flipflops, only those flipflops with the following features will be accessed.

- the last flipflop in a flipflop chain that has a fanout to all non-flipflop/latch gates,
- all the flipflops in a flipflop chain that have fanouts to all non-flipflop/latch gates,

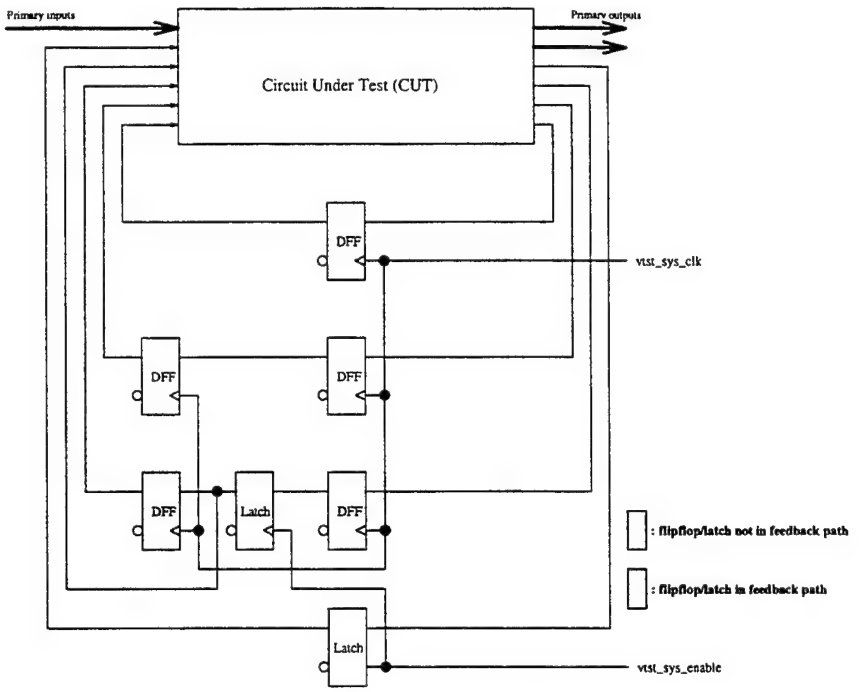


Figure 6.1: Sequential circuit

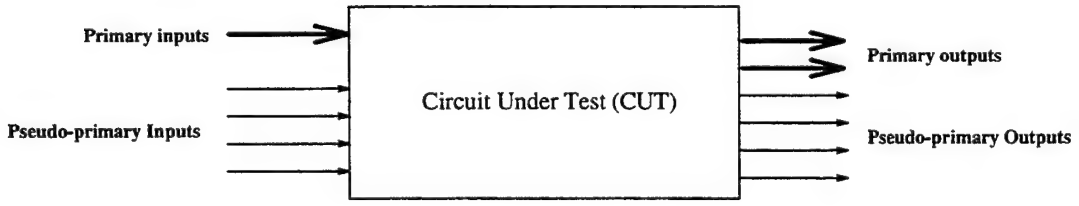


Figure 6.2: Pseudo-combinational circuit for test purpose

6.1 Pseudo-Combinational Circuit and Pseudo-Scan Design

Using a scan path to scan in the test patterns and then scan out the responses has been widely used and is known as *scan design*. Chapter 9 will discuss the scan design developed in VTST. There is another approach that will access the internal nodes using a pseudo-random approach without actually scanning test patterns into each scan flipflop and scanning out each response. It is the idea of *pseudo-scan testing*. It is a methodology that converts a sequential circuit into a combinational circuit during the test mode which we called it pseudo-combinational circuit. For example, the sequential circuit shown in Figure 6.1 can be converted to a pseudo-combinational circuit for test purposes as shown in Figure 6.2.

Instead of scanning in each test pattern to the internal flipflops and then applying the pattern to the CUT via these flipflops, the notion of pseudo-scan testing is to apply test pattern generated from an LFSR to the fanouts of the internal flipflops via Mux's added. As shown in the Figure 6.3, the flipflops that qualify based on the rules stated above will

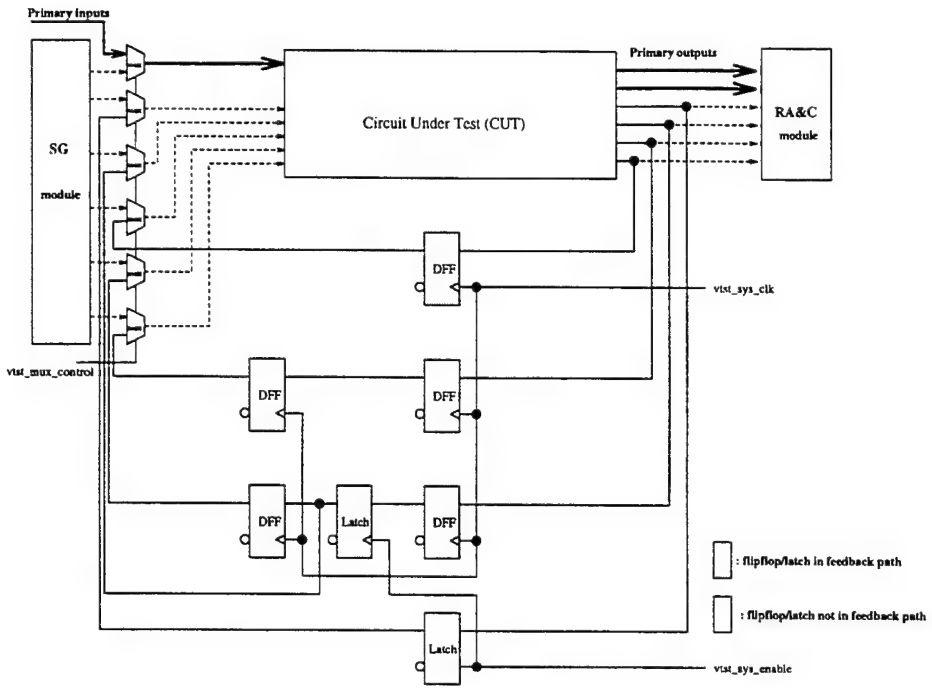


Figure 6.3: Pseudo-scan testing

have MUXes added to their fanouts and the MUXes will direct the patterns from the SG module to the CUT in test mode and direct the output of the internal flipflops to the CUT during the normal operation. On the output side, the inputs to the first flipflops in flipflop chains will be observed by the RA&C module as primary outputs.

There are three different design methodologies for pseudo-scan design. They are:

- Pseudo-scan without flipflops and latches, which we label as *RegFree*,
- Pseudo-scan without flipflops and all latches in feedback path, which we label as *FF-Free*,
- Pseudo-scan without all flipflops and latches in feedback path, which we label as *FbcFree*.

6.2 Pseudo-Scan Algorithm

The algorithm used to perform pseudo-scan without any flipflops and latches is:

1. find all the flipflops and latches that have no fanins and fanouts to any other flipflops or latches, these flipflops and latches are called *single flipflops* and *single latches*,
2. find all the flipflop chains and latch chains, these flipflops and latches are called *chained flipflops* and *chained latches*,
3. for each single flipflop or latch, a pseudo-primary input and a pseudo-primary output will be added. To add a pseudo-primary input, a MUX is added and connected to

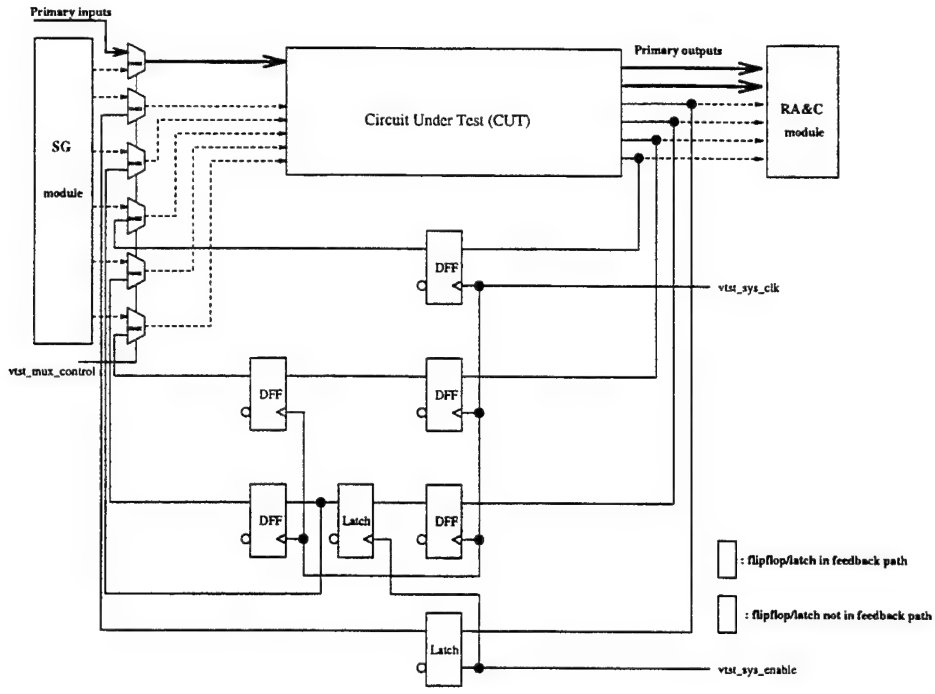


Figure 6.4: Pseudo-scan without flip-flops and latches

the single flipflop fanout. A pseudo-primary input is added to one of the inputs of the MUX and the output port of the single flipflop will be connected to another input port of the MUX.

4. for each flipflop-chain or latch-chain, a pseudo-primary output will be added but more than one pseudo- primary input will be added.
   - for the last chained flipflop or latch, a pseudo-primary input is added via a MUX.
   - for each chained flipflop or latch except the last one, a pseudo-primary input will be added if there is at least one fanout of that chained flipflop or latch that is not a flipflop or latch.

The pseudo-scan algorithm can be applied to the three methodologies with some minor modification.

6.3 Pseudo-Scan without Flip-flops and Latches

In this design methodology, all the flipflops and latches will be isolated from the CUT during testing. Pseudo-primary inputs will be added via MUXes and connected to the SG module. Figure 6.4 depicts this pseudo-scan design method. The pseudo-scan algorithm can be applied to this design methodology without any modification.

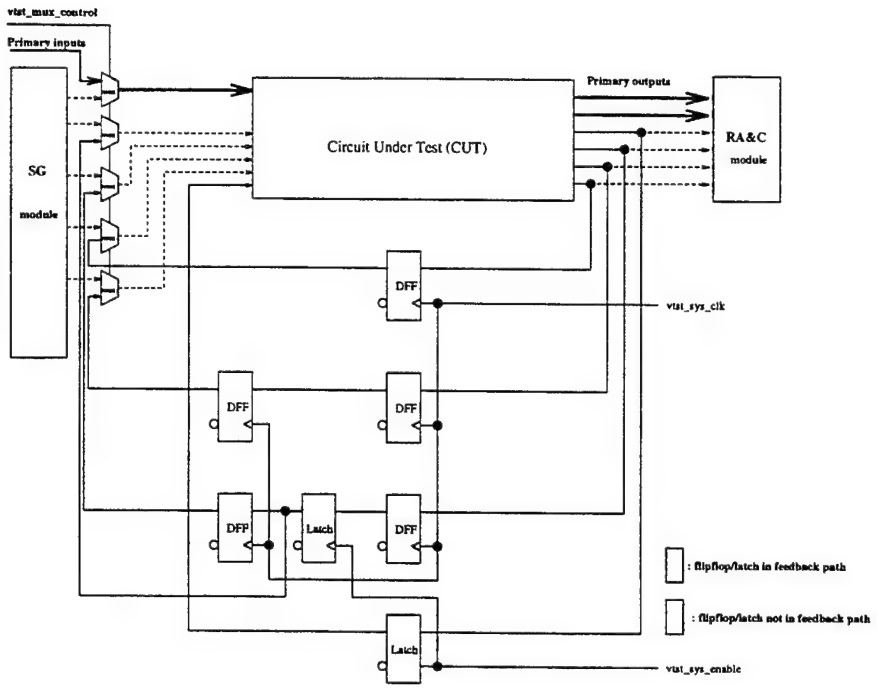


Figure 6.5: Pseudo-scan without flip-flops and latches in the feedback path

6.4 Pseudo-Scan without Flip-Flops and No Latches in the Feedback Path

In this design methodology, all the flipflops and any latch that are in feedback paths will be isolated from the CUT during testing. Pseudo-primary inputs will be added via Mux's and be connected to the SG module. As shown in Figure 6.5, pseudo-primary inputs will be added via Mux's and be connected to the SG module if the latches and flipflops are not in the feedback path. Therefore, the portion of the circuit with the non-feedback flipflops and latches is unmodified. But 4 pseudo-primary inputs have been added via 4 Mux's and are connected to the SG module.

The pseudo-scan algorithm for this design methodology will have the following modification:

1. find all the flipflops and latches that have no fanins and fanouts to any other flipflops/latches or any latches that are not in feedback paths; these flipflops and latches are called *single flipflops* and *single latches*,

2. find all the flipflop chains and latch chains; these flipflops and latches are called *chained flipflops* and *chained latches*. If any latch that is not in a feedback path but is in the midst of the flipflop-chain or latch-chain, the flipflop-chain or latch-chain becomes two or more chains. And each chain is considered separately for pseudo-primary input/output insertion.

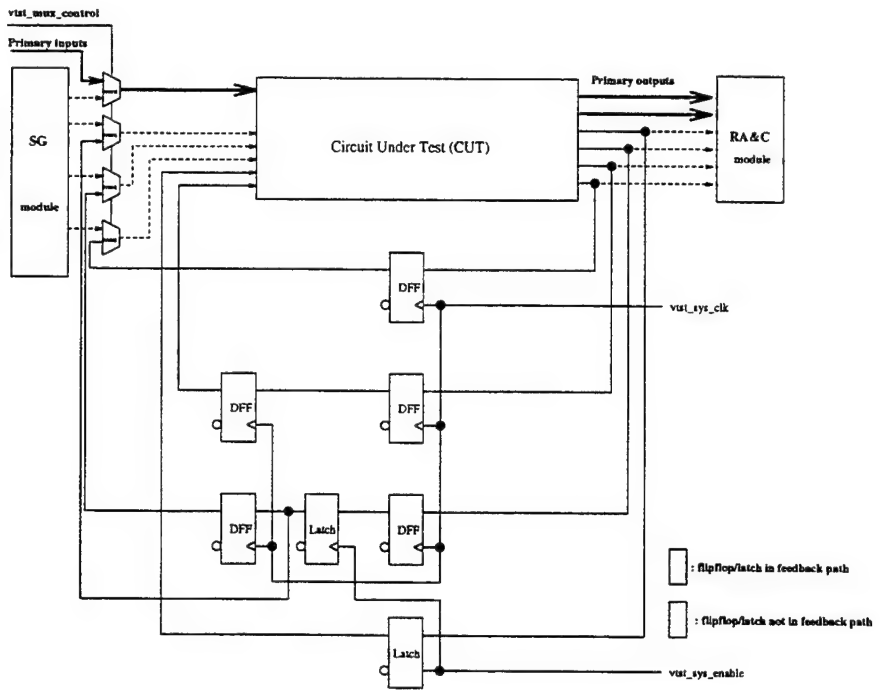


Figure 6.6: Pseudo-scan with flip-flops and latches in the feedback path

6.5 Pseudo-Scan without Flip-flops and latches in the Feedback Path

In this design methodology, all the flipflops and latches that are in feedback paths will be isolated from the CUT during testing. Pseudo-primary inputs will be added via Mux's and be connected to the SG module. As shown in Figure 6.6, pseudo-primary inputs will be added via Mux's and be connected to the SG module if the latches and flipflops are not in the feedback path. Therefore, the portion of the circuit with the non-feedback flipflops and latches is unmodified. But 3 pseudo-primary inputs have been added via 3 Mux's and are connected to the SG module.

The pseudo-scan algorithm for this design methodology will have the following modification:

1. find all the flipflops and latches that have no fanins and fanouts to any other flipflops/latches or any flipflop/latches that are not in feedback paths; these flipflops and latches are called *single flipflops* and *single latches*,

2. find all the flipflop chains and latch chains; these flipflops and latches are called *chained flipflops* and *chained latches*. If any flipflop/latch that is not in a feedback path but in the midst of the flipflop-chain or latch-chain, the flipflop-chain or latch-chain becomes two or more chains. And each chain is considered separately for pseudo-primary input/output insertion.

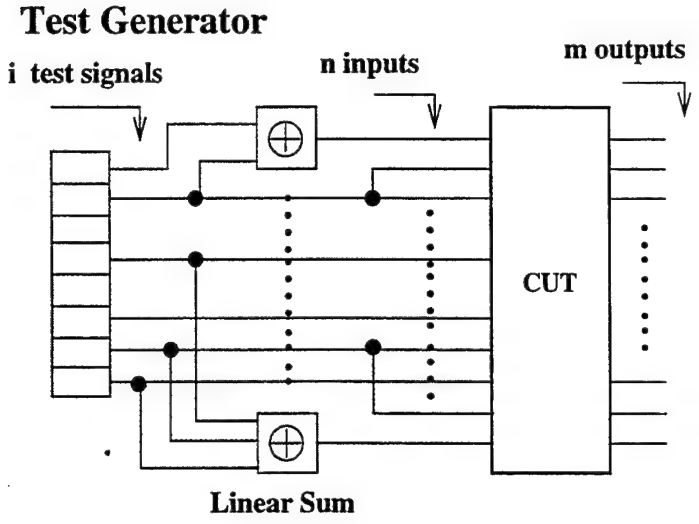
Chapter 7

Automated Synthesis Tool for Pseudo-Exhaustive Test Generator: BISTSYN

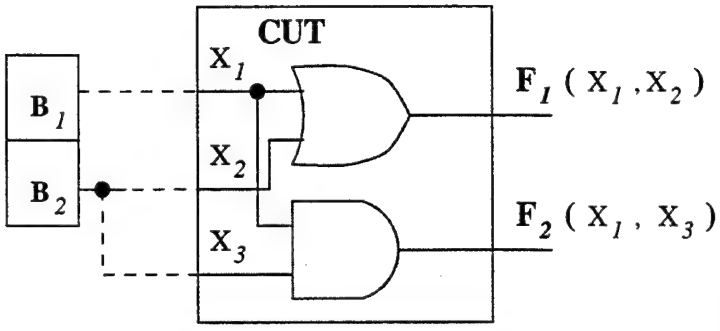
Built-In Self Test (BIST) has been proposed as a powerful technique for addressing the highly complex testing problems of VLSI circuits. In the BIST methodology, two major problems which must be addressed are test generation and response analysis. In this chapter, we present an efficient unified procedure, named Three-Phase cluster partitioning, to automatically synthesize a pseudo-exhaustive test generator for VLSI BIST design. Previous approaches to the problem of test generation have optimized computational efficiency at the expense of the required hardware overhead, or vice-versa. Our design procedure is computationally efficient and produces test generation circuitry with low hardware overhead. The procedure minimizes the number of test patterns that are required for pseudo-exhaustive test. Based on Three-Phase Cluster Partitioning, a design generator named BISTSYN has been developed and implemented to facilitate the BIST design. The input to the design generator is a circuit description at the gate level which is viewed as a netlist. BISTSYN provides the BIST mechanisms as the output. For those conventional circuits which are extremely unsuitable for pseudo-exhaustive test, BISTSYN employs a circuit partitioning tool, named Autonomous, to partition the combinational portion of the circuit into different structural subcircuits so that each subcircuit can be pseudo-exhaustively tested. We demonstrate the effectiveness of BISTSYN by applying the method to different examples and practical VLSI designs. The detailed comparisons of our benchmark simulation results against those that would be obtained by existing techniques are also presented.

7.1 Introduction

Testing a circuit consists of applying an input sequence and observing the resulting output sequence. With respect to test generation, the central problem in today's IC technology



(a)

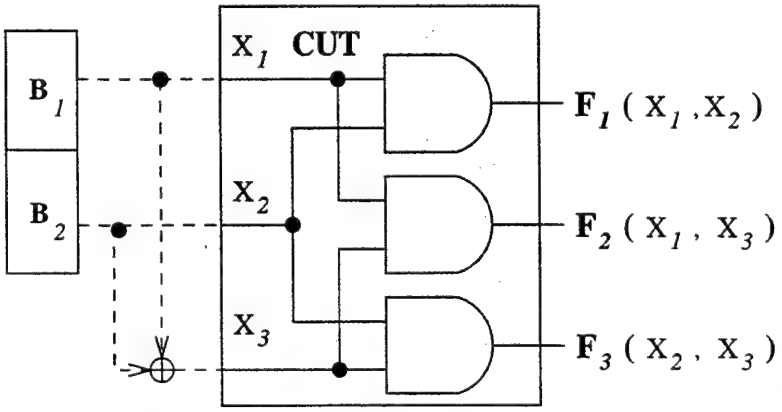


X_1	X_2	X_3
0	0	0
0	1	1
1	0	0
1	1	1

Pseudo-exhaustive test set

(b)

Note: ---- test signal flow



X_1	X_2	X_3
0	0	0
0	1	1
1	0	1
1	1	0

Pseudo-exhaustive test set

(c)

Note: ---- test signal flow

Figure 7.1: Testing scheme by BISTSYN

is that a highly complex chip has low accessibility of internal circuit nodes (controllability and observability). The low accessibility makes traditional testing techniques costly and ineffective. Especially, modern VLSI processors incorporate hundreds of thousands of transistors to perform the bulk of the task of a complete computer system on a single chip. This level of integration has substantially increased the complexity of testing such that classical functional based testing methods are simply inadequate. Thus, design for testability techniques are in demand.

Built-In Self-Test (BIST) has been proposed as a powerful design for testability technique for addressing the highly complex VLSI testing problems [1, 2, 3, 4, 5, 6, 7, 8, 9]. BIST design includes on-chip circuitry to provide test patterns and to analyze the output response. It can perform the testing internal to the chip so that the need for complex external testing equipment is greatly reduced. Using BIST, the test volume can be significantly reduced, and many of the traditional testing problems mentioned above can be overcome. BIST techniques can be classified into two categories, namely *on-line* BIST and *off-line* BIST. In *on-line* BIST, testing occurs during normal functional operation, i.e., the circuit under test (CUT) is not placed into a test mode while normal functional operation is locked out. In *off-line* BIST, the CUT is tested while it is not carrying out its normal function. This form of testing can be applicable at the system, board, and chip level. Often *off-line* testing is carried out using on-chip or on-board test pattern generators and output response analyzers. This chapter deals primarily with *off-line* BIST.

Pseudo-exhaustive test is a BIST design methodology that provides 100-percent fault coverage for all testable stuck-at faults without the need for fault simulation or deterministic test generation. Pseudo-exhaustive test is based on the observation that even though the number of primary inputs to a CUT may be prohibitively large, most primary outputs of such a circuit often functionally depend on a relatively small number of these inputs. Formally, an output is said to be functionally dependent on an input if and only if there exists a path in the circuit from the input to the output. For such a circuit, pseudo-exhaustive test can be performed for each output by applying all the combinations to only those inputs upon which the output functionally depends. Moreover, the same test signal can be applied to two input lines of a circuit concurrently if none of the outputs is functionally dependent on both of the inputs. These facts can be used to reduce the required number of test signals and to perform the pseudo-exhaustive test of several circuit outputs concurrently.

The first method for pseudo-exhaustive test pattern generation was proposed in [10] wherein a syndrome drive counter (SDC) designed by a minimum covering procedure was proposed. The technique searches for the different inputs of a CUT that can share the same test signal. But, test time may be still too long when only a few inputs share the same test signal. Furthermore, no attempt is made to find the minimum number of test signals for the non-MTC circuits.

If the number of required test signals is equal to the maximum number of inputs upon which any output depends, the circuit is called an MTC (Maximum Test Concurrency) circuit [11].

Two universal testing techniques have been proposed to reduce the number of test signals for pseudo-exhaustive test. The first technique is the verification testing method [11] which uses constant weight counters (CWC) to implement pseudo-exhaustive test generators. A major problem with this approach is that input stimulus generation must be computed, which is an NP-complete problem. For circuits having a higher order m-out-of-n code (the set of codewords of n binary bits where each codeword has exactly m 1's) [11], CWC is very costly to implement. The second technique is to use a combination of linear feedback shift registers (LFSRs) and exclusive-or (XOR) gates [12, 13]. However, the techniques proposed in these papers did not attempt to allocate the minimum number of required test signals, nor did they attempt to minimize the extra hardware overhead (i.e. XOR gates). Since these problems are also NP-complete, it is unlikely that an explicit formula can be found to calculate the exact minimum number of the required test signals. A universal procedure (LFSRs/XORs) proposed in [12] provides an upper bound of the required test signals for the CUT. However, both of the universal testing techniques neither keep track of the specific dependency of each output nor provide the detailed method for minimizing the number of required test signals, and simply focus on w, the maximum number of inputs upon which any output depends. Therefore, these techniques derive pseudo-exhaustive tests for any output having the dependency set with the size equal to or less than w. However, in general, not all outputs depend on the same number, w, of inputs. Therefore, the number of test patterns found in both techniques can be more than what is required for pseudo-exhaustive test.

Condensed LFSR testing based on linear codes and structural partitioning was proposed for self-testing in [14]. This technique is most effective when $w \geq \frac{n}{2}$ (n is the number of inputs). However, when $w < \frac{n}{2}$, more test patterns than necessary will be generated.

Another design technique using a LFSR to generate pseudo-exhaustive test patterns was proposed in [15]. The technique is based on cyclic codes which are easy to implement and have less hardware overhead than a LFSR designed using general linear codes. An (n, k) cyclic code over the Galois field of 2 contains a set of 2^k distinct codewords, each of which is an n-tuple satisfying the following property: if c is a codeword, then the n-tuple obtained by rotating c one bit to the right is also a code word. Cyclic codes are a subclass of linear codes. However, more test patterns than necessary may be generated in some cases. For example, to design an (n, k) LFSR for an (n, w) CUT, we need to find a cyclic code from Appendix D in [16] that has a code length n and a minimum distance, $w + 1$. However, such a cyclic code may not exist. If a cyclic code for code length n does not exst, then we need to find a cyclic code which has next higher code length ($> n$). If the minimum distance $w + 1$ does not exist, then find a cyclic code with next higher distance ($> w + 1$). In these worse cases, the test patterns generated by this cyclic code generator may be more than what is required for pseudo-exhaustive test.

In this chapter, an efficient algorithm, *Three-Phase Cluster Partitioning,* is presented to automatically design a pseudo-exhaustive test generator for BIST. The algorithm generates fewer test signals compared to existing techniques and is suitable for both MTC and non-MTC circuits. The test generation procedure operates in three phases. The first phase consists of a cluster partitioning technique that itself is sufficient for circuits having

the MTC property. The second and third phases further reduce the required number of test patterns for non-MTC circuits by forming linear sum of the test signals obtained at the end of the first phase. The basic test scheme is shown in Figure 7.1(a) where we use i test signals instead of n ($i < n$) for the test pattern generator (the i test signals can be generated by a LFSR or a counter) to generate all pseudo-exhaustive test patterns. Figure 7.1(b), for example, shows how the outputs of a 3-input and 2-output CUT can be pseudo-exhaustively tested with just two test signals. The output F_1 functionally depends on x_1 and x_2 only. Similarly, the output F_2 functionally depends on x_1 and x_3 only. Since neither of the two outputs functionally depend on both x_2 and x_3, one may connect x_2 and x_3 together and apply all four pseudo-exhaustive tests to the CUT as shown in Figure 7.1(b). Now, $F_1(x_1, x_2)$ is exhaustively tested and so is $F_2(x_1, x_3)$. Hence, in this case, pseudo-exhaustive test has reduced the number of tests from eight (exhaustive) to four (pseudo-exhaustive).

Let us consider the circuit shown in Figure 7.1(c). Each output depends on two inputs. Hence, the above scheme can not be used. However, one can easily verify that four tests are sufficient to pseudo-exhaustively test the circuit. Note that for each of the four tests, $x_3 = x_1 \oplus x_2$. This example shows that the linear sum can be used to reduce the number of tests required for pseudo-exhaustive test.

The main idea of the proposed approach is to minimize the number of test signals, i, for pseudo-exhaustive test and also to minimize the number of the XOR gates used in the formation of linear sum. For those conventional circuits which are extremely unsuitable for pseudo-exhaustive test (the number of test signals is prohibitively large), *BISTSYN* employs a circuit partitioning tool, named *Autonomous* [17], to partition the circuit into different subcircuits, each of which can then be pseudo-exhaustively tested. In fact, it is necessary to have each partitioned subcircuit with the property that each output is functionally dependent on a sufficiently small number of inputs. Access to the embedded inputs and outputs of each subcircuit can be achieved by inserting multiplexers (MUXs) and connecting the embedded inputs and outputs of each subcircuits to those primary inputs and outputs that are not used by the subcircuits.

The basic scheme of partitioning a circuit into two subcircuits by *Autonomous* is demonstrated by the example shown in Figure 7.2. In Figure 7.2(a), a combinational logic circuit is partitioned into two subcircuits, C_1 and C_2. Many such partitions exist for large scale, multiple-output circuits. Figure 7.2(b) depicts how the multiplexers are inserted between two subcircuits, where A' and C' represent a subset of the signals in A and C and are used in testing C_2 and C_1 respectively. By controlling the multiplexers, all the inputs and outputs of each subcircuit can be accessed using primary input and output lines. For example, to test subcircuit C_1 the multiplexers can be controlled as depicted in Figure 7.2(c) to completely access all the inputs and outputs. The partitioning scheme enhances the controllability and observability of inputs and outputs associated with C_1 and C_2. When used with *Autonomous* on a set of benchmark examples, *BISTSYN* has been found to lead to a BIST design which achieves a higher fault coverage in comparison with previous work.

The remainder of this chapter is organized as follows. In Sections 7.2,7.3 and 7.4, *Three-Phase Cluster Partitioning* algorithm will be introduced. The steps of the algorithm will be illustrated through detailed examples. In Section 7.5, the results of applying the *BIST-SYN* to benchmark circuits and different VLSI designs will be given. Section 7.6 contains conclusions and suggestions for future work.

7.2 Phase One: Cluster Partitioning

Many researchers have observed that even though the number of primary inputs to a CUT may be prohibitively large, the primary output of such a circuit often depends on a relatively small number of these inputs. Figure 7.3 exhibits ten benchmark circuits [18]. For the five ALU and Control circuits, it is found that even if the number of primary inputs is large, only one of these five circuits is functionally dependent on all the primary inputs. So, the same test signal can be applied to two input lines of a CUT if none of the outputs is functionally dependent on both of the inputs. This fact can be used to reduce the required number of test signals for such circuits.

The problem of systematically reducing the number of test signals in this fashion while still exhaustively testing the CUT has been formulated as a covering problem of the cliques of a graph [10]. However, the problem of finding the cliques of a graph, a *maximal* complete sub-graph of the original graph, is NP-complete. Thus, an efficient heuristic algorithm is needed to obtain a practical implementation for large problems.

Before we discuss the phase one cluster partitioning algorithm in detail, some terms used in [10] need to be reviewed:

Definition 1: An input pair (x_i, x_j) is said to be *adjacent* if there exists at least one output function f_i, $i = 1, 2, ..., m$ which functionally depends on both inputs x_i and x_j.

Definition 2: A pair (x_i, x_j) is said to be *non-adjacent* if they are not *adjacent,* i.e. input lines x_i and x_j can be tested by the same test signal.

Definition 3: A *non-adjacent (NA)* graph is defined to have n vertices, designated by x_1, x_2, ..., x_n corresponding to the n input lines. There is an edge between node x_i and node x_j if and only if the pair (x_i, x_j) is *non-adjacent.*

Definition 4: Two circuit input lines x_i, x_j which can be tested by a common test signal is represented in NA graph by a composite node. In other words, we replace the two nodes x_i and x_j and the edge joining them with a single composite node labeled as [x_i, x_j].

As in Ref. [10], a NA graph is used to determine which sets of circuit input lines can be connected to common test signals. Specifically, a set of input lines can all be connected to the same test signal if they form a *clique* in NA graph. Therefore, their objective is to find the minimum number of disjoint cliques in NA graph. A graph is called *complete* if all of

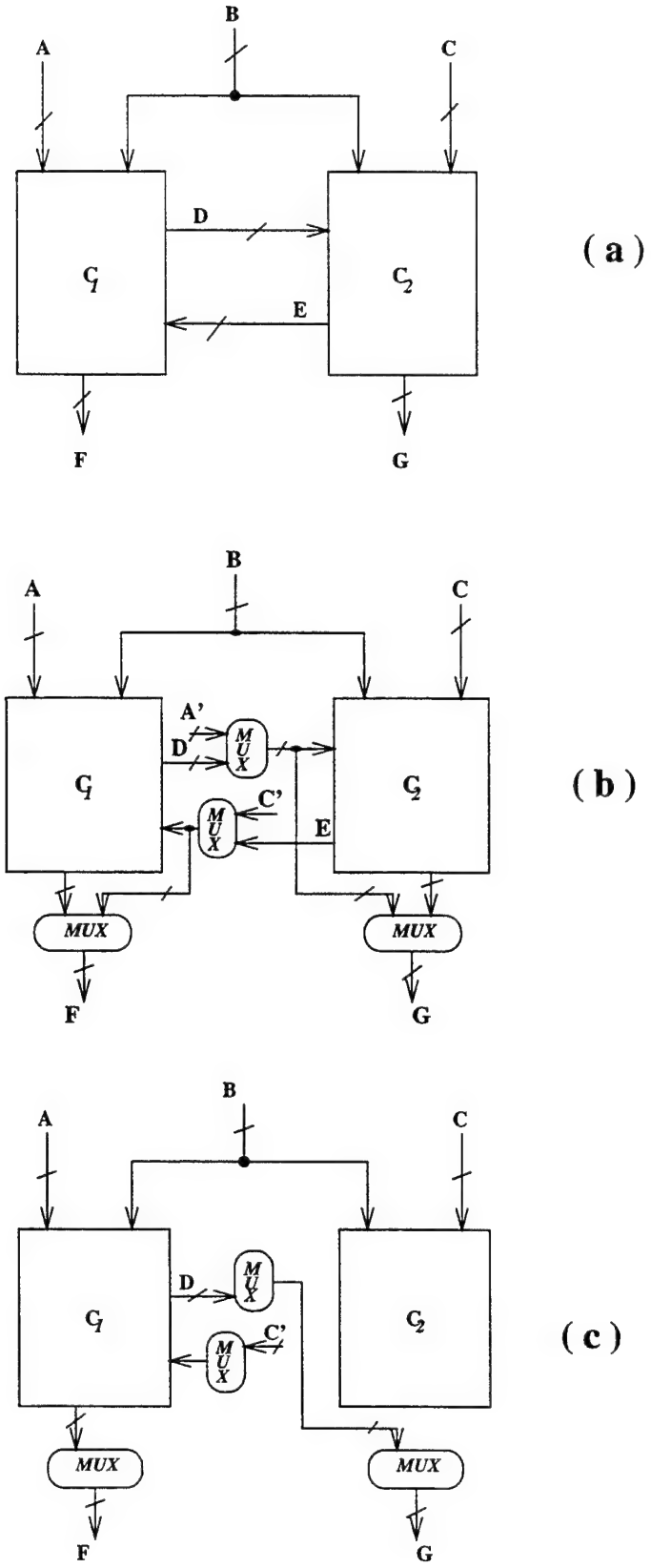


Figure 7.2: Network partition scheme

Circuit	Function	Total Gates	Inputs Lines	Output Lines	Max. Dependency
C432	Priority Decoder	160	36	7	36
C499	ECAT	202	41	32	41
C880	ALU and Control	383	60	26	45
C1355	ECAT	546	41	32	41
C1908	ECAT	880	33	25	33
C2670	ALU and Control	1,193	233	140	122
C3540	ALU and Control	1,669	50	22	50
C5315	ALU and Selection	2,307	178	123	67
C6288	16-bit Multiplier	2,406	32	32	32
C7552	ALU and Control	3,512	207	108	194

Figure 7.3: Ten ISCAS benchmark combinational circuits

its vertices are connected to all of the other vertices in the graph. A *clique* is a maximal complete sub-graph of a graph [25]. In other words, a clique is a complete sub-graph that is not contained within any other subgraph that is also complete. A cluster is a complete sub-graph but not necessarily a maximal complete sub-graph. It should be pointed out that the resulting clusters obtained from our algorithm are not necessarily all cliques, as the cliques of a graph are not required to be disjoint. (See for example the cliques of the Moon-Moser graphs as given in [25].) We prefer the term "cluster partitioning" for this reason. Each cluster is a complete sub-graph representing a test signal and all the nodes (input lines) of each cluster are tested by a common test signal. Further investigation of the cluster partitioning algorithm has shown it generates a smaller number of disjoint clusters compared to the clique partitioning algorithm for large scale graphs. So, our objective is to find the minimum number of disjoint clusters that will cover all the nodes of NA graph. In this chapter, we exploit the "neighborhood property" in a graph to obtain a polynomial-time procedure.

Definition 5: A node x_k in NA graph is said to be a common neighbor of an edge $(x_i,\ x_j)$ if edges exist from x_k to both x_i and x_j.

Definition 6: A deleted edge of a composite node $[\ x_i,\ x_j\ ]$ can arise in any of the following three ways:

(1) A node x_k which is not a common neighbor of x_i and x_j will no longer be connected to the composite node $[\ x_i,\ x_j\ ]$. Thus, the edge $(x_i,\ x_k)$ or $(x_j,\ x_k)$ has to be deleted, as appropriate.

(2) A node x_k which is a common neighbor of x_i and x_j will still be connected to the composite node $[\ x_i,\ x_j\ ]$. However, only one of the original two edges needs to be retained. Thus, one of the edges $(x_i,\ x_k)$ or $(x_j,\ x_k)$ has to be deleted. In this algorithm, the edge $(x_j,\ x_k)$ will be deleted if $j\ >\ i$. Otherwise, the edge $(x_i,\ x_k)$ will be deleted.

(3) The edge $(x_i,\ x_j)$ itself must, of course, be deleted.

Definition 7: A candidate pair $(x_u,\ x_v)$ in NA graph is determined by the following criteria (listed in order of priority):

(1) $(x_u,\ x_v)$ has the minimum number of deleted edges.

(2) $(x_u,\ x_v)$ has the maximum number of common neighbors.

The cluster partitioning algorithm can now be stated in the following way:

Step 1: *Establish the NA graph using the functional dependency sets Fi of the m outputs of the CUT.*

Step 2: *Traverse the list of edges in the NA graph. For each edge $(x_a,\ x_b)$, compute the number of common neighbors and the number of deleted edges.*

Step 3: *Find the candidate pair $(x_u,\ x_v)$. Choose the smaller of p and q as the head of the cluster. Remove the deleted edges of the composite node $[x_u,\ x_v]$. Update the list of edges accordingly and recompute the numbers of common neighbors and deleted edges. If the list of edges is empty, then cluster partitioning is complete.*

Step 4: *Assume x_u is the head of the current cluster. Find a candidate pair which joins node x_u and another node, x_r. Choose the smaller of u and r as the head of the resulting cluster. Remove the deleted edges of the composite node $[x_u,\ x_r]$. Update the list of edges accordingly and recompute the numbers of common neighbors and deleted edges. If the list of edges is empty, then cluster partitioning is complete. Otherwise, if node x_u (or x_r if $r < u$) no longer appears in the updated list of edges, go to step 3 and start to form another cluster. Otherwise, repeat step 4 and continue to find other nodes to be added to the current cluster.*

The implemented pseudo code of phase one algorithm can now be described in the following way:

Phase_One_Algorithm();

m : the number of outputs of the CUT;

n: the number of edges in NA graph;

E: e_1, e_2,..., e_n where e_i is an edge in NA graph;

C: c_1, c_2,..., c_n where c_i is number of common neighbors of e_i;

D: d_1, d_2,..., d_n where d_i is number of deleted edges of the composite node $[x_a,x_b]$ where x_a and x_b have the edge e_i connecting them before the composite node $[x_a,\ x_b]$ is formed;

F : F_1, F_2, ..., F_m where F_i is the functional dependency set of the i-th output function;
Cluster is a set of nodes that can share the same test signals;

x_H: head of cluster;

Procedure *Compute_common_neighbor_and_deleted_edge(n, E, C, D)*

C = NULL; D = NULL;

for i = 1 to n compute c_i; $C = C \cup c_i$; compute d_i; $D = D \cup d_i$;

return(C, D);

Procedure *Find_candidate_pair_and_update_edge_list(n, E, C, D, x_H)*
if (x_H! = *NULL*) then $x_u = x_H$;
With C and D, find candidate pair (x_u, x_v);
t = min$u$, v;
$x_H = x_t$; /** use the node with the smallest numerical value as the head of cluster **/
E_d = deleted edges of (x_u, x_v);
n_d = number of deleted edges of (x_u, x_v);
n = n - n_d; E = E - E_d; *Compute_common_neighbor_and_deleted_edge(n, E, C, D)*;
return(C, D, x_H);
begin /** Phase_One_Algorithm starts **/
establish NA graph using the functional dependency sets in F ;
Compute_common_neighbor_and_deleted_edge(n, E, C, D);
while(E != NULL) /** Formation of cluster **/
x_H = NULL; *Find_candidate_pair_and_update_edge_list(n, E, C, D, x_H);*
while(E != NULL) /** Include other nodes to the current cluster **/
Find_candidate_pair_and_update_edge_list(n, E, C, D, x_H);
if(E == NULL or x_H no longer appears in any edge of E) then
break; /** Exit the loop to form a new cluster **/
end; /** *Phase_One_Algorithm* ends **/

Assume p is the number of disjoint clusters partitioned by the phase one algorithm. Then p is the number of required test signals, i.e. the test set $T = \{s_1, s_2, ..., s_p\}$. Each element of T is composed of the CUT inputs which have been formed into a cluster. All the CUT inputs in the cluster s_i are represented by the same symbol x_j, which is the input in the cluster having the smallest numerical value of j, and the functional dependency sets are then updated accordingly. If w, the maximum number of inputs upon which any output depends is equal to p (i.e., the MTC situation) then there is no need to proceed with phase two and phase three procedures. Otherwise, the phase two and phase three algorithms, described in the following Sections respectively, must be subsequently applied to obtain an optimized test generation hardware.

The following example illustrates how the phase one algorithm is applied to a CUT.

Example 1: An 8-input, 5-output CUT is given in Figure 7.4(a). The output functional dependency sets, $F_i's$, are listed as follows:

$$F_1 = \{x_1, x_2, x_3, x_4\}$$
$$F_2 = \{x_6, x_7, x_8\}$$
$$F_3 = \{x_3, x_4, x_5, x_6\}$$
$$F_4 = \{x_3, x_5, x_6, x_7\}$$
$$F_5 = \{x_1, x_4, x_7, x_8\}$$

The NA graph established by these dependency sets $F_i's$ is shown in Figure 7.4(b). The list of edges in the NA graph, the number of deleted edges and the number of common neighbors are shown in Figure 7.5. When this initial list is examined, it is found that two pairs (x_1, x_6) and (x_3, x_8) have the same minimum number (=3) of deleted edges and the same number (=0) of common neighbors. So, we arbitrarily choose the pair (x_1, x_6) as the candidate pair. Accordingly, *node* x_1 becomes the head of the resulting cluster. The

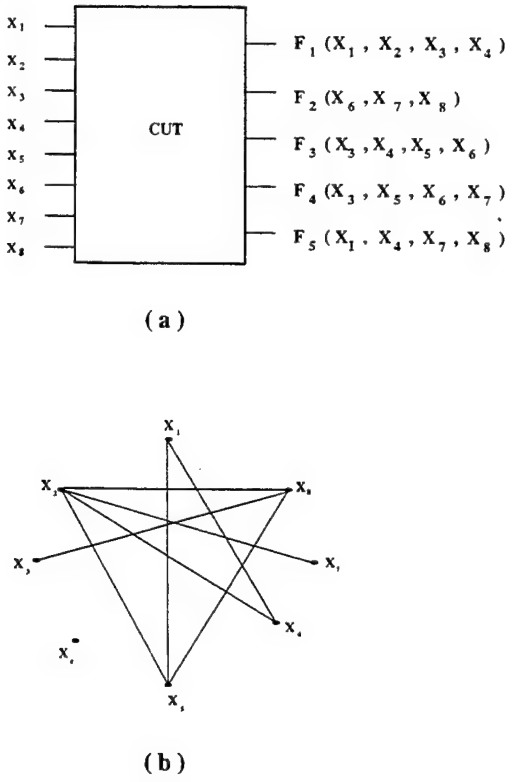


Figure 7.4: Circuit and NA graph of Example 1

deleted edges (x_1, x_5), (x_1, x_6) and (x_2, x_6) of the composite node $[x_1, x_6]$ are removed. Figure 7.6 tabulates the list of edges and their associated numbers of deleted edges and common neighbors. Since *node* x_1 no longer appears in the list of edges, the phase one algorithm starts to form another cluster. In Figure 7.6, four edges (x_2, x_5), (x_2, x_7), (x_3, x_8) and (x_5, x_8) have the same minimum number (=3) of deleted edges. Among them, the edges (x_2, x_5) and (x_5, x_8) have the same maximum number (=1) of common neighbors, so we select (x_2, x_5) as the next candidate pair. *Node* x_2 becomes the head of the resulting cluster. The deleted edges (x_2, x_5), (x_2, x_7) and (x_5, x_8) of the composite node $[x_2, x_5]$ are removed accordingly.

In Figure 7.7, since *node* x_2 still appears in the list, we continue to pick a candidate pair which joins *node* x_2 to other nodes. Thus, the pair (x_2, x_8) is selected as the next candidate pair. Finally, after combining the pair (x_2, x_8), the list of edges is empty. The phase one algorithm is complete.

The algorithm partitions the input pins into five clusters (i.e. five test signals). They are $\{(s_1), (s_2), (s_3), (s_4), (s_5)\} = \{(x_1, x_6), (x_2, x_5, x_8), (x_3), (x_4), (x_7)\}$. Therefore, $p = 5$ and the test set $T = \{ x_1, x_2, x_3, x_4, x_7 \}$. The updated functional dependency sets are listed as follows.

	The list of edges	Number of Deleted Edges	Number of Common Neighbors
candidate pair	(1,5)	4	0
---->	(1,6)	3	0
	(2,5)	5	1
	(2,6)	5	0
	(2,7)	4	0
	(2,8)	5	1
	(3,8)	3	0
	(5,8)	4	1

Figure 7.5: Selection of first candidate pair

$$F_1 = \{x_1, x_2, x_3, x_4\}$$
$$F_2 = \{x_1, x_2, x_7\}$$
$$F_3 = \{x_1, x_2, x_3, x_4\}$$
$$F_4 = \{x_1, x_2, x_3, x_7\}$$
$$F_5 = \{x_1, x_2, x_4, x_7\}$$

Using the phase one algorithm, we find that five test signals can pseudo-exhaustively test the circuit. The test scheme is shown in Figure 7.8. But we know $w = 4$ and $p \neq w$. The circuit is a non-MTC circuit. Therefore, we would continue using the phase two algorithm to reduce the number of required test signals. After applying phase two algorithm to this non-MTC circuit, four test signals are found and they are sufficient for pseudo-exhaustively testing the CUT. The steps of the phase two algorithm applied to example 1 will be illustrated in the next Section.

We have tested the phase one algorithm on several very large circuits. In Ref. [18],

	The list of edges	Number of Deleted Edges	Number of Common Neighbors
candidate pair			
---->	(2,5)	3	1
	(2,7)	3	0
	(2,8)	4	1
	(3,8)	3	0
	(5,8)	3	1

Figure 7.6: Selection of second candidate pair

candidate pair	The list of edges	Number of Deleted Edges	Number of Common Neighbors
---->	(2,8)	2	0
	(3,8)	2	0

Figure 7.7: Selection of third candidate pair

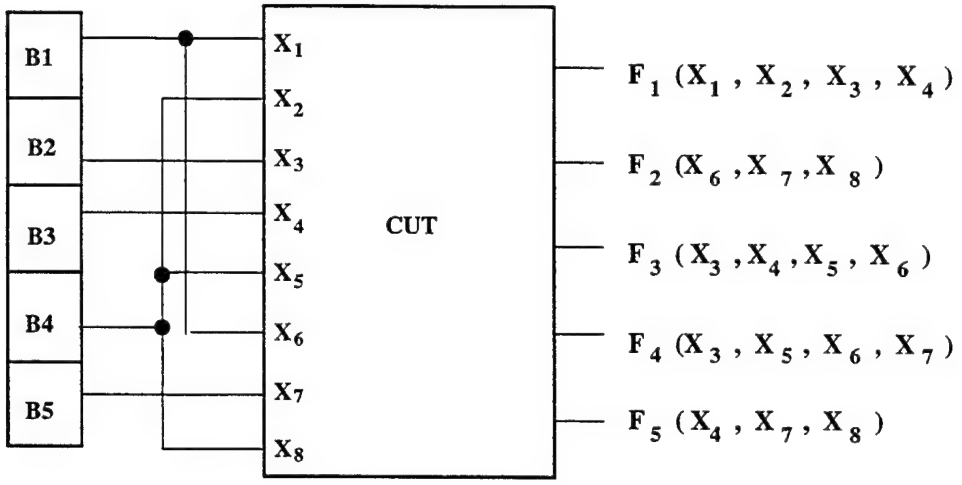


Figure 7.8: First test scheme of Example 1

several large circuits have been proposed as benchmarks for test generation algorithms. Of these, the circuits C880, C2670, C5315 and C7552 have the property that no primary output depends on all the primary inputs, and thus pseudo-exhaustive test techniques can be used. After applying the phase one algorithm to these four circuits, we find that all of them are MTC circuits. The number of test signals that are required for pseudo-exhaustively testing each of these circuits is described in Figure 7.9. The computation times (on a

Circuit Name	Max. Dependency	Phase One: Cluster Partitioning		Clique Partitioning	
		Number of Test Signals	Execution Time(sec)	Number of Test Signals	Execution Time(sec)
C880	45	45	1	45	4
C2670	122	122	99	122	145
C5315	67	67	150	68	223
C7552	194	194	14	194	16

Figure 7.9: Results of 4 Benchmark circuits by phase one algorithm

Sun SparcStation 10) required to obtain these results are also listed in the table. These circuits contain up to several thousand gates and several hundred input/output lines, yet the computation time in all cases is quite reasonable. Comparable results using the clique partitioning are also shown in Figure 7.9. As can be seen, application of cluster partitioning algorithm yields excellent results for these large circuits.

7.3 Phase Two: Formation of Linear Sum of Two Test Signals

For non-MTC circuits, the number of required test signals can be further reduced by using linear combinations of a smaller number of signals. Akers has proposed a linear sum approach for the test generation problem [12]. However, our approach differs from his in three respects. First, we do not begin to form linear sum until reductions in the number of test signals by the phase one procedure have been achieved. Second, our approach aims at finding the minimal number of exclusive-or (XOR) gates which are added. Third, our approach searches the design space for forming the linear sum of two test signals first. If this formation is not found then the approach starts the *phase three* procedure (described in next Section) to search for the formation of linear sum of more than two test signals. These three factors usually result in significant savings in both hardware and number of test patterns that are required to pseudo-exhaustively test non-MTC circuits.

The phase two algorithm is listed as follows. Note that in the phase two algorithm, the procedure implementing the phase three algorithm (see details in Section 7.4) is called when the formation of linear sum of more than two test signals is requested.

Step 1: *From the functional dependency sets F_i, arbitrarily choose one set F_d which has exactly w elements ($w < p$) i.e. $F_d = \{ t_1, t_2, \ldots, t_w \}$, where $t_i \in T$. Then form the exclusive set $E = T - F_d$ i.e. $E = \{e_1, e_2, \ldots, e_k \}$ where $k = p - w$. Set an index $l = 1$.*

Step 2: *Traverse the functional dependency sets and find those which contain the element e_l. Let P_j denote such a set. Each set $P_j - \{e_l, e_l + 1, \ldots, e_k \}$ contains adjacent nodes, so that these sets can be used to construct an adjacency graph. Establish the corresponding non-adjacency graph NA_l from this graph.*

Step 3:

(i) If $l < p - w$, choose a cluster C_l of size two in the NA_l graph which has the largest number of elements that are also members of the previous clusters C_k, $k < l$ from the graphs NA_k. (Note that these clusters are not necessarily disjoint since they are obtained from different NA graphs.) If there is more than one such cluster, then choose the cluster which will result in the maximum number of edges in the NA_{l+1} graph. (In this step, the algorithm attempts to explore the design space for the formation of linear sum for the rest of inputs $e_{l+1}, \ldots, e_k$ in E that have not been formed linear sum.)

(ii) If $l = p - w$, then arbitrarily choose any cluster C_l of size two in the NA_l graph.

(iii) If there are no edges in the NA_l graph, then go to the phase three procedure to check whether there exists a test signal for the input pin e_l by formation of a linear sum of three or more signals in F_d. If it exists, then we replace the test signal for e_l by XOR

of the found signals in the phase three procedure and update the functional dependency sets accordingly and go to step 5. Otherwise, we set $F_d = F_d \cup e_l$ *and go to step 5.*

Step 4: *Replace the test signal for* e_l *by the XOR of the two signals in the cluster found in the previous step. Update the functional dependency sets accordingly.*

Step 5: *If* $l = p - w$*, then the phase two algorithm is complete and the size of* F_d *is the number of required test signals. Otherwise, set* $l = l + 1$ *and go to step 2.*

The implemented pseudo code of phase two algorithm is described as follows.

Phase_Two_Algorithm()
m : the number of outputs of the CUT;
p : the number of test signals obtained after *Phase_One_Algorithm()*
w : the maximum number of inputs upon which the outputs depend;
F : F_1, F_2, ..., F_m where F_i is the functional dependency set of the i-th output function;
T : the set of test signals obtained after Phase_One_Algorithm();
F_d: a functional dependency set having w elements, i.e. $F_d = t_1, t_2, ..., t_w$ where $t_i \in T$ and $p < w$;
begin /** Phase Two Algorithm begin **/
arbitrarily choose F_d where $F_d \in F$;
form the exclusive set $E = T - F_d$ where E = e_1, e_2, ..., e_k, k = p - w;
for ($l = 1$ to $p - w$)
/** for loops begin **/
$j = 0$; for(i= 1 to m) if($e_l \in F_i$) then $j++$; form the set P_j - e_l, e_{l+1}, ..., e_k;
jj=j; establish NA_l graph based on the sets P_j - e_l, e_{l+1}, ..., e_k's where $j = 1, ..., jj$;
if (there is at least one edge in NA_l graph) then
if ($l < p - w$) then
choose a cluster C_l of size two in NA_l graph which has the largest number of elements that are also members of the previous clusters C'_l, $l' < l$ from the graphs NA'_l;
/** These clusters aren't necessarily disjoint as they're obtained from different NA graphs **/
if (l = 1 or there is more than one such cluster) then
choose the cluster which will result in maximum number of edges in NA_{l+1};
/** the algorithm attempts to explore the design space for the formation of linear sum for the rest of inputs e_{l+1}, ..., e_k in E that have not been formed linear sum **/
else arbitrarily choose any cluster C_l of size two in NA_l graph;
$e_l = t_i \oplus t_j$ where $C_l = t_i$, t_j and t_i, $t_j \in F_d$; update all F_i accordingly;
/** if ends **/
else /** there is no edge in NA_l graph **/
linear_sum_exist =
Phase_Three_Algorithm(e_l, F_d);
if(linear_sum_exist == $TRUE$) then
$e_l = t'_1 \oplus t'_2 \oplus t'_3 ... \oplus t'_i$ where t'_1, t'_2, t'_3, ..., $t'_i \in F_d$; update all F_i accordingly;
else $F_d = F_d \cup e_l$;
/** else ends **/

/** for loop ends **/
end; /** *Phase_Two_Algorithm* ends **/
Phase_Three_Algorithm(e_l, F_d)
size: number of elements in F_d or user specifies if number of elements in F_d is too large;
begin /** *Phase_Three_Algorithm* starts **/
linear_sum_exist = $FALSE$;
for(i=2 to size)
if(linear sum of i elements of F_d is formed for e_l) then
/** $e_l = t'_1 \oplus t'_2 \ldots \oplus t'_i$ where $t'_1, t'_2, \ldots, t'_i \in F_d$; verified by Theorem 2 **/
linear_sum_exist = $TRUE$; break;
return(linear_sum_exist);
end; /** *Phase_Three_Algorithm* ends **/

Lemma 1:
If $|F_d| > w$, then a LFSR sequence with length $2^{|F_d|} - 1$ can pseudo-exhaustively test the CUT. Otherwise, $|F_d| = w$, and we need a LFSR sequence with length $2^{|F_d|}$ including all-zero $|F_d|$-tuple (called de Bruijn sequence) to pseudo-exhaustively test the CUT.

Theorem 1:
A possible test signal for the input pin e_l can be the linear sum of test signals x_u and x_v, if x_u and x_v form an edge in NA_l graph.

Proof: In phase two algorithm, all the sets P_j's containing the element e_l are selected from the functional dependency sets. The term $P_j - \{e_l, e_{l+1}, \ldots\ldots, e_k\}$ defines a set of adjacent nodes which can be formed into an adjacency graph. Note that the NA graph is the dual graph of the adjacency graph. Thus, there does not exist any set P_j which contains all the elements x_u, x_v and e_l, if x_u and x_v form an edge in the NA_l graph. In this case, the test signal for input pin e_l can be the linear sum of test signals x_u and x_v.

The following is an application of the phase two algorithm to the non-MTC circuit of example 1. The steps are applied to the five clusters produced by phase one.

We arbitrarily choose $F_d = F_1 = \{ x_1, x_2, x_3, x_4 \}$. The exclusive set $E \equiv T - F_d$ i.e. $E = x_7$. Then, set $l = 1$. Next, traverse the functional dependency sets $F_i's$ (i =1 to 5) to find all the sets $P_1 = F_2$, $P_2 = F_4$ and $P_3 = F_5$, where $P_j's$ contain the element x_7. The NA_1 graph is established by the following sets and is shown in Figure 7.10.

$P_1 - x_7 = \{ x_1, x_2 \}$
$P_2 - x_7 = \{x_1, x_2, x_3 \}$
$P_3 - x_7 = \{x_1, x_2, x_4 \}$
From the NA_1 graph, $C_1 = \{x_3, x_4 \}$ is the only cluster found with a size of 2. Since $C_1 \neq \emptyset$, we continue to execute the phase two algorithm. The test signal for the input pin x_7 can be the XOR of the test signals for the input pins x_3 and x_4 (i.e. $x_7 = x_3 \oplus x_4$). Substituting x_3 and x_4 for x_7 in all functional dependency sets $F_i's$, we have $F_1 = F_2 = F_3 = F_4 = \{ x_1, x_2, x_3, x_4 \}$. Since $l = 1 = p - w$, the phase two algorithm is complete. The size of F_d is four. In other words, four test signals are sufficient

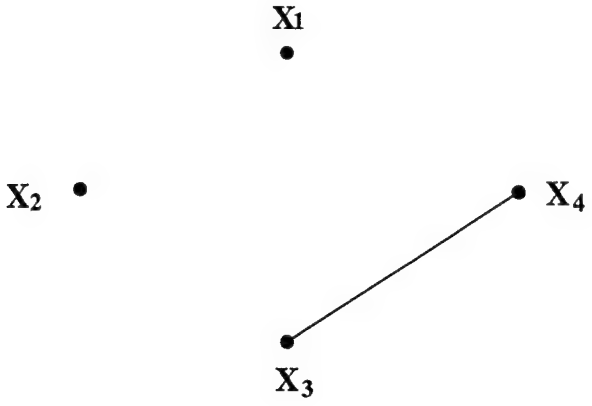


Figure 7.10: NA_1 graph of example 1

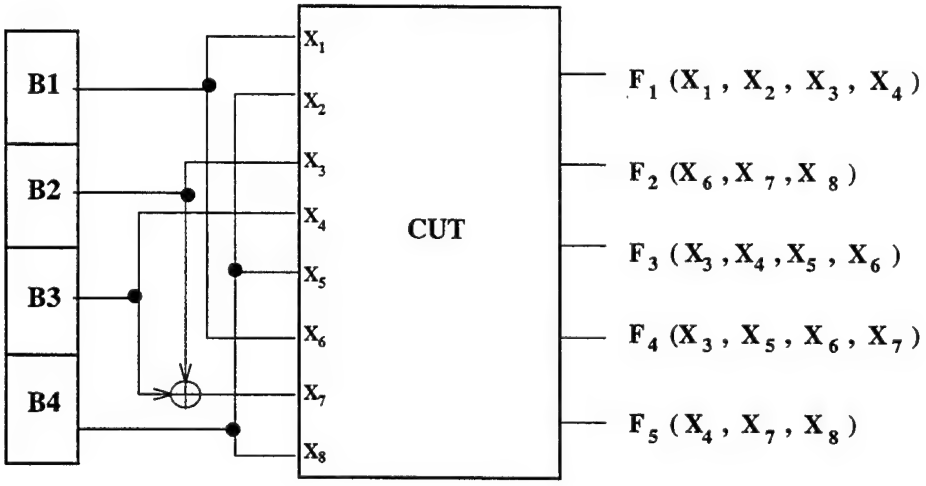


Figure 7.11: Second test scheme of example 1

to pseudo-exhaustively test the circuit. The testing scheme is shown in Figure 7.11.

7.4 Phase Three: Formation of Linear Sum of Multiple Test Signals

In the previous Section, the proposed techniques are limited to the formation of linear sum of two test signals. However, if the linear sum of two test signals does not exist, then instead of adding a new test signal for the test input, the formation of linear sum of multiple test signals should be investigated. In this Section, the problem of formation of linear sum of multiple test signals is formulated as a mathematical problem. Assume $F_d = t_1, t_2, ..., t_w$ and we try to express the test signal e_l not in F_d as a linear sum of elements in F_d in such a way that all functional dependency sets F_i are linearly independent.

Let V be the set of all linear combinations (linear sum) of F_d. Then V forms a w-

dimensional vector space over the finite field 0,1. Every vector in V then admits a raw vector representation. Thus, if $v = r_1t_1 \oplus r_2t_2 \oplus ..... \oplus r_wt_w$ is in V, then its row vector representation is $[r_1, r_2, ..., r_w]$.

Let $S = \{v_1, v_2, ..., v_m\}$ be a subset of V. If $m > w$, then V will be linearly dependent. If $m \leq w$, then we can test the linear dependence of S in the following way. Let $v_i = r_{i,1}t_1 \oplus r_{i,2}t_2 \oplus ..... \oplus r_{i,w}t_w$, so that it has row vector representation $[r_{i,1}, r_{i,2}, ..., r_{i,w}]$, $i = 1, 2, ..., m$. From these row vectors, we form the $m \times w$ matrix

$$M_S = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,w} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,w} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ r_{m,1} & r_{m,2} & \cdots & r_{m,w} \end{bmatrix}$$

We recall the following definition in linear algebra. The row rank of a matrix M_S is the maximum number of linearly independent rows of M_S and the column rank of a matrix M_S is the maximum number of linearly independent columns of M_S. A result in linear algebra tells us that the row rank and column rank of any matrix are the same. So the set S is linearly independent if and only if the row rank of M_S is m, i.e., if and only if the column rank of M_S is m. Suppose that the column rank of M_S is m. This means that we can find m columns of M_S which form a linearly independent set. In other words, the $m \times m$ submatrix of M_S formed from these columns must be non-singular, i.e., its determinant must be non-zero. The converse of this also holds. So if we can find an $m \times m$ submatrix of M_S which has non-zero determinant, then its column rank is m. Summarizing, we have the following theorem for linear independence:

Theorem 2: The output functional dependency set F_i is linearly independent if and only if the $m \times w$ matrix M_S, where $S = F_i$, has an $m \times m$ submatrix with a non-zero determinant, where $|F_i| = m$, $|F_d| = w$, and $m \leq w$.

Proof: Let $F_d = t_1, t_2, \ldots, t_w$ and $F_i = v_1, v_2, \ldots, v_m$ where $v_i = r_{i,1}t_1 \oplus ri_2t_2 \oplus \ldots \oplus r_{i,w}t_w$. Therefore, we form a $m \times w$ matrix

$$M_{F_i} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,w} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,w} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ r_{m,1} & r_{m,2} & \cdots & r_{m,w} \end{bmatrix}$$

to express F_i in the w-dimensional vector space V over the finite field 0,1. V is the set of all linear combinations (linear sum) of F_d. As discussed before, if M_{F_i} has an $m \times w$ submatrix with a non-zero determinant then the row rank of M_{F_i} is m and v_1, v_2, ..., v_m are linear independent. Therefore, the output functional dependency sets F_i is linear independent and vice versa.

The following is an example of circuit in which the phase three procedure is applied to further reduce the number of required test signals for pseudo-exhaustive test.

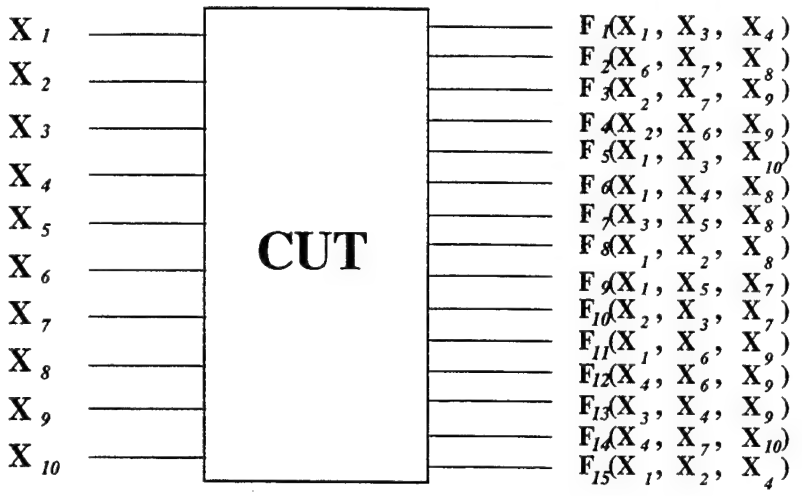


Figure 7.12: Circuit example 2

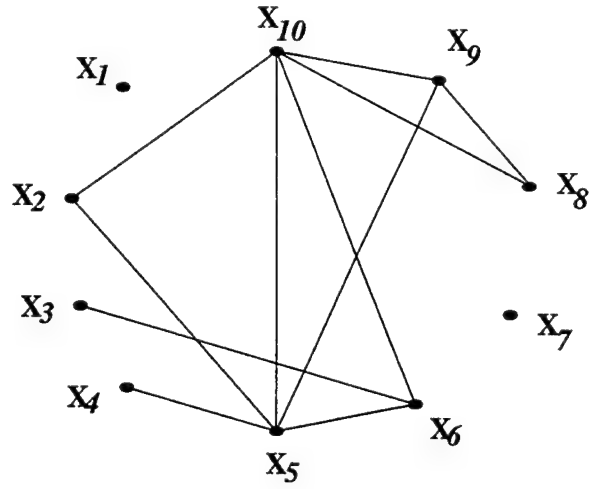


Figure 7.13: NA graph of example 2

Example 2: A 10-input, 15-output CUT is given in Figure 7.12, which is more suitable to most of pseudo-exhaustive techniques described in the Introduction Section since each output function is functionally dependent on the same number (=3) of its primary inputs. According to the phase one algorithm, the NA graph of the circuit is established as shown in Figure 7.13. When the phase one algorithm is complete, the final clusters are $\{(x_1), (x_2), (x_3, x_6), (x_4, x_5), (x_7), (x_8, x_9, x_10)\}$. Therefore, $p = 6$. The test set is $T = x_1, x_2, x_3, x_4, x_7, x_8$. All the functional dependency sets are updated and listed as follows.

$F_1 = \{ x_1, x_3, x_4 \}$, $F_2 = \{ x_3, x_7, x_8 \}$ $F_3 = \{ x_2, x_7, x_8 \}$ $F_4 = \{ x_2, x_3, x_8 \}$
$F_5 = \{ x_1, x_3, x_8 \}$ $F_6 = \{ x_1, x_4, x_8 \}$ $F_7 = \{ x_3, x_4, x_8 \}$ $F_8 = \{ x_1, x_2, x_8 \}$
$F_9 = \{ x_1, x_4, x_7 \}$ $F_{10} = \{ x_2, x_3, x_7 \}$ $F_{11} = \{ x_1, x_3, x_8 \}$ $F_{12} = \{ x_3, x_4, x_8 \}$
$F_{13} = \{ x_3, x_4, x_8 \}$ $F_{14} = \{ x_4, x_7, x_8 \}$ $F_{15} = \{ x_1, x_2, x_4 \}$

But, $w = 3$ and $p \neq w$; the circuit is therefore a non-MTC circuit. So, the phase

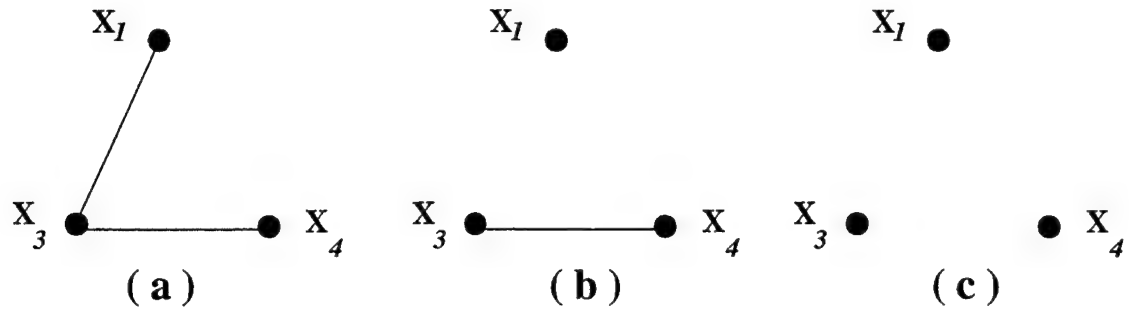


Figure 7.14: NA graphs for Example 2

two algorithm is used. Initially, we choose $F_d = F_1 = \{ x_1, x_3, x_4 \}$. The *exclusive set* $E \equiv T - F_d$ and $E = \{ x_2, x_7, x_8 \}$. So, $k = 3$, $e_1 = x_2$, $e_2 = x_7$ and $e_3 = x_8$. Then, set $l = 1$. Traverse the functional dependency sets $F_i's$ (i =1 to 15) and find all the sets $P_1 = F_3$, $P_2 = F_4$, $P_3 = F_8$, $P_4 = F_{10}$ and $P_5 = F_{15}$, where $P_j's$ contain the element $e_1 = x_2$. The NA_1 graph is established by the following sets and shown in Figure 7.14(a).

$P_1 - \{ x_2, x_7, x_8 \} = \emptyset$
$P_2 - \{ x_2, x_7, x_8 \} = x_3$
$P_3 - \{ x_2, x_7, x_8 \} = x_1$
$P_4 - \{ x_2, x_7, x_8 \} = x_3$
$P_5 - \{ x_2, x_7, x_8 \} = \{ x_1, x_4 \}$.

From the NA_1 graph, we choose $C_1 = \{ x_1, x_3 \}$. The test signal for input pin x_2 can be the XOR of the test signals for input pins x_1 and x_3 (i.e. $x_2 = x_1 \oplus x_3$). Substitute x_1 and x_3 for x_2 in all functional dependency sets $F_i's$. Therefore,

$F_1 = \{ x_1, x_3, x_4 \}$ $F_2 = \{ x_3, x_7, x_8 \}$ $F_3 = \{ x_1, x_3, x_7, x_8 \}$ $F_4 = \{ x_1, x_3, x_8 \}$
$F_5 = \{ x_1, x_3, x_8 \}$ $F_6 = \{ x_1, x_4, x_8 \}$ $F_7 = \{ x_3, x_4, x_8 \}$ $F_8 = \{ x_1, x_3, x_8 \}$
$F_9 = \{ x_1, x_4, x_7 \}$ $F_{10} = \{ x_1, x_3, x_7 \}$ $F_{11} = \{ x_1, x_3, x_8 \}$ $F_{12} = \{ x_3, x_4, x_8 \}$
$F_{13} = \{ x_3, x_4, x_8 \}$ $F_{14} = \{ x_4, x_7, x_8 \}$ $F_{15} = \{ x_1, x_3, x_4 \}$

Since $l = 1 \neq p - w$, phase two algorithm is not complete. We set $l = 2$. Next, We traverse the functional dependency sets $F_i's$ (i =1 to 15) and find all the sets $P_1 = F_2$, $P_2 = F_3$, $P_3 = F_9$, $P_4 = F_{10}$ and $P_5 = F_{14}$, where the $P_j's$ contain the element $e_2 = x_7$. The NA_2 graph is established by the following sets and shown in Figure 7.14(b).

$P_1 - \{ x_7, x_8 \} = x_3$
$P_2 - \{ x_7, x_8 \} = x_1, x_3$
$P_3 - \{ x_7, x_8 \} = x_1, x_4$
$P_4 - \{ x_7, x_8 \} = x_1, x_3$
$P_5 - \{ x_7, x_8 \} = x_4$

From the NA_2 graph, $C_2 = \{ x_3, x_4 \}$ is the only cluster found with a size of 2. The test signal for input pin x_7 can be the XOR of the test signals for input pins x_3 and x_4 (i.e. $x_7 = x_3 \oplus x_4$). Substitute x_3 and x_4 for x_7 in all functional dependency sets $F_i's$.

Therefore,

$F_1 = \{ x_1, x_3, x_4 \}$ $F_2 = \{ x_3, x_4, x_8 \}$ $F_3 = \{ x_1, x_3, x_4, x_8 \}$ $F_4 = \{ x_1, x_3, x_8 \}$
$F_5 = \{ x_1, x_3, x_8 \}$ $F_6 = \{ x_1, x_4, x_8 \}$ $F_7 = \{ x_3, x_4, x_8 \}$ $F_8 = \{ x_1, x_3, x_8 \}$
$F_9 = \{ x_1, x_3, x_4 \}$ $F_{10} = \{ x_1, x_3, x_4 \}$ $F_{11} = \{ x_1, x_3, x_8 \}$ $F_{12} = \{ x_3, x_4, x_8 \}$
$F_{13} = \{ x_3, x_4, x_8 \}$ $F_{14} = \{ x_3, x_4, x_8 \}$ $F_{15} = \{ x_1, x_3, x_4 \}$

Again, $l = 2 \neq p - w$, the phase two algorithm is not complete. We set $l = 3$ and then traverse the functional dependency sets $F_i's$ (i =1 to 15) and find all the sets

$P_1 = F_2$, $P_2 = F_3$, $P_3 = F_4$, $P_4 = F_5$, $P_5 = F_6$, $P_6 = F_7$, $P_7 = F_8$, $P_8 = F_{11}$, $P_9 = F_{12}$, $P_10 = F_{13}$ and $P_11 = F_{14}$, where the $P_j's$ contain the element $e_3 = x_8$. The NA_3 graph can be established by the sets $P_j - x_8$ and is shown in Figure 7.14(c). We find that there are no edges in NA_3 graph and $C_3 = \emptyset$. The test signal for input pin x_8 cannot be a linear sum of any two signals in the current test set F_d (= x_1, x_3, x_4). Without applying the phase three procedure to consider the linear sum of multiple test signals, an extra test signal x_8 is added to the current test set F_d, i.e., $F_d = F_d \cup x_8 = x_1, x_3, x_4, x_8$ and the phase two algorithm is complete. But, this design procedure leaves a better solution out of consideration and generates a non-optimal solution. In the phase three procedure, while traversing the test signals in F_d, we find that the test signal for input pin x_8 can be a linear sum of the test signals for input pins x_1, x_3 and x_4 (i.e. $x_8 = x_1 \oplus x_3 \oplus x_4$). This can be verified in Figures 7.15(a) and (b) where $w = m = 3$. As shown in Table 7.15(b) all M_{F_i}, $i = 1, ..., 15$, are 3×3 matrices with non-zero determinant. Therefore, by Theorem 2, all the output functional dependency sets are linearly independent. The formation of linear sum for x_8 is valid. No extra test signal is needed to add to the basic test set F_d. Then, we substitute x_1, x_3 and x_4 for x_8 in all functional dependency sets.

Since $l = 3 = p - w$, so both the phase two and three algorithms are complete. The size of F_d is three. Therefore, three test signals (8 test patterns) are sufficient to pseudo-exhaustively test the circuit. The test generator is shown in Figure 7.16.

For completeness, in Figure 7.17(a), we also give the results produced by five previously proposed test generation methods ("SDC"[10], "LFSRs/XORs"[12], "CWC"[11], "Condensed LFSR"[14] and "LFSR with Cyclic Code"[15]) for the circuits of examples 1 and 2. For the circuit of example 1, using the "SDC" technique will require $p = 5$ test signals that will generate $2^5 - 1 = 31$ test patterns; using the "LFSRs/XORs" technique will require 5 test signals (= 31 test patterns) that is computed by the equation in [12] where the number of test signals $I(n, w) = I(8, 4) = 1 + I(7, 3) = 1 + \log_2 7 + 1 = 5$; Using the "CWC" will require $U(p, w) = U(5, 4) = 2^4 = 16$ test patterns (refer to the equation in [11]) ; Using the "Condensed LFSR" technique will require 2^k ($= 2^4 = 16$) test patterns where $w = 4$, $p = 5$, and $k = 4$ is the smallest positive integer with $w \leq [\frac{k}{p-k+1}] + [\frac{k}{p-k+1}]$ (refer to the equation in [14]); Using the "LFSR with Cyclic Code" technique will require 63 test patterns (refer to Table 3 in [15]). For the circuit of example 2, using the "SDC" technique will require $p = 6$ test signals that is $2^6 - 1 = 63$ test patterns; using the "LFSRs/XORs" technique will require 5 test signals ($= 2^5 - 1 = 31$ test patterns) that is

(a)

$F_1(X_1, X_3, X_4)$
$F_2(X_3, X_3 \oplus X_4, X_1 \oplus X_3 \oplus X_4)$
$F_3(X_1 \oplus X_3, X_3 \oplus X_4, X_1 \oplus X_3 \oplus X_4)$
$F_4(X_1 \oplus X_3, X_3, X_1 \oplus X_3 \oplus X_4)$
$F_5(X_1, X_3, X_1 \oplus X_3 \oplus X_4)$
$F_6(X_1, X_4, X_1 \oplus X_3 \oplus X_4)$
$F_7(X_3, X_4, X_1 \oplus X_3 \oplus X_4)$
$F_8(X_1, X_1 \oplus X_3, X_1 \oplus X_3 \oplus X_4)$
$F_9(X_1, X_4, X_3 \oplus X_4)$
$F_{10}(X_1 \oplus X_3, X_3, X_3 \oplus X_4)$
$F_{11}(X_1, X_3, X_1 \oplus X_3 \oplus X_4)$
$F_{12}(X_3, X_4, X_1 \oplus X_3 \oplus X_4)$
$F_{13}(X_3, X_4, X_1 \oplus X_3 \oplus X_4)$
$F_{14}(X_4, X_3 \oplus X_4, X_1 \oplus X_3 \oplus X_4)$
$F_{15}(X_1, X_1 \oplus X_3, X_4)$

(b)

$$M_{F_1} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad M_{F_2} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_3} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_4} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_5} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$M_{F_6} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_7} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_8} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_9} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad M_{F_{10}} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

$$M_{F_{11}} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_{12}} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_{13}} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_{14}} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad M_{F_{15}} = \begin{matrix} X_1\ X_3\ X_4 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 7.15: Formation of linear sum in output dependency sets of example 2

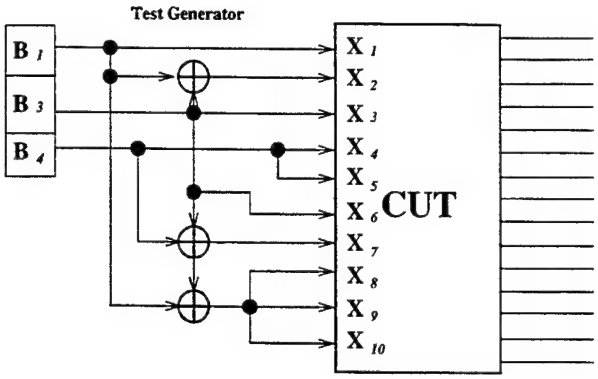


Figure 7.16: Test Generator of Example 2

computed by the equation $I(n,w) = I(15,3) = [\log_2 15] + 1 = 5$; Using the "CWC" will require $U(p,w) = U(6,3) = 2 \times p = 12$ test patterns (refer to Table VIII in [11]); Using the "Condensed LFSR" technique will require $2^k - 1$ $(= 2^4 - 1 = 15)$ test patterns where $w = 3$, $p = 6$, and $k = 4$ is the smallest positive integer with $w \leq [\frac{k}{p-k+1}] + [\frac{k}{p-k+1}]$; Using the "LFSR with Cyclic Code" technique will require 15 test patterns.

Finally, the flow chart of *Three-Phase Cluster Partitioning* algorithm is described in Figure 7.18. Figure 7.19 depicts the flow chart of the BISTSYN system. The circuit analyzer embedded in BISTSYN will perform the ATPG. The ATPG results such as the test patterns, the fault coverage and a list of untestable and dropped faults will then be passed to the user. Next, the circuit analyzer will generate a graph representation for the CUT for the system use.

Three Phase Cluster Partitioning algorithm will take the graph representation and automatically generate a pseudo-exhaustive test generator. If the user satisfies the number of pseudo-exhaustive test patterns, then the system exits. No fault simulation is required since all testable faults are detected by the pseudo-exhaustive test patterns. Otherwise, the pseudo-random test will be performed based on the reduced size pseudo-exhaustive test generator wherein the fault coverage of the circuit is increased more rapidly with every random pattern applied to the circuit, and also the number of test patterns required as compared to the non-reduced case is highly reduced. Referring to the fault simulations of C2670 shown in Figure 7.20, the test patterns generated by the reduced size LFSR (122-bit) achieve a much higher fault coverage than the same amount of test patterns generated by the non-reduced size LFSR (233-bit). Moreover, the 122-bit LFSR achieves the same fault coverage more rapidly than the 233-bit LFSR.

For example, to achieve 88% fault coverage, the 122-bit LFSR requires about 200,000 test patterns, but the 233-bit LFSR requires about 500,000 test patterns. During performing the pseudo-random test, the pseudo-exhaustive test generator will generate test patterns based on the test length specified by the user. Through the fault simulator, the user can evaluate the fault coverage. During each fault simulation, the test length is increased and the fault list will be updated accordingly. If the fault coverage and the test

Circuits	SDC	LFSR/XOR	CWC	Condensed LFSR	LFSR with Cyclic Code	BISTSYN
Example 1	31	31	16	16	63	8
Example 2	63	31	12	31	15	8

(a)

Number of Inputs	Number of Outputs	W	P	SDC	LFSR/XOR	CWC	Condensed LFSR	LFSR with Cyclic Code	BISTSYN
5	5	3	3	8	15	8	8	8	8
5	5	4	4	16	16	16	16	63	16
5	5	5	5	32	32	32	32	63	32
8	8	2	2	4	15	4	4	7	4
8	8	3	4	15	15	8	15	8	8
8	8	4	4	16	31	16	16	63	16
8	8	5	5	32	63	32	32	63	32
8	8	6	6	64	127	64	64	64	64
8	8	7	8	255	128	128	255	2,047	128
10	10	2	2	4	15	4	4	7	4
10	10	3	3	8	15	8	8	8	8
10	10	4	4	31	63	16	16	63	16
10	10	5	5	32	63	32	32	63	32
10	10	6	6	127	127	64	64	64	64
10	10	7	7	511	255	170	255	2,047	128
10	10	9	9	1,023	512	512	1,023	2,047	512
10	10	10	10	1,024	1,024	1,024	1,024	2,047	1,024
10	15	2	3	7	15	4	4	7	4
10	15	3	4	15	31	8	8	8	8
10	15	4	5	31	63	16	16	63	16
10	15	5	7	127	63	42	63	63	32
10	15	6	9	511	127	120	255	1,023	64
10	15	7	9	511	255	170	255	2,047	128
10	15	8	9	511	511	256	256	2,047	256
10	15	9	10	1,023	512	512	512	2,047	512
15	10	2	2	4	15	4	4	7	4
15	10	3	3	8	31	8	8	8	8
15	10	4	4	16	63	16	16	63	16
15	10	5	5	32	127	32	32	63	32
15	10	6	6	64	255	64	64	64	64
15	10	7	7	2,047	511	330	1,023	1,047	128
15	10	9	11	2,047	2,047	682	1,023	2,047	512
15	10	10	12	4,095	2,047	1,365	2,047	2,047	1,024
15	15	2	3	7	15	4	4	7	4
15	15	3	4	15	31	8	8	8	8
15	15	4	4	16	63	16	16	63	16
15	15	5	6	63	127	32	32	63	32
*15	15	6	8	255	255	36	127	1,023	64
15	15	7	13	8,191	511	572	2,047	1,023	255
15	15	8	9	511	1,023	256	256	2,047	256
15	15	9	12	4,095	2,047	992	2,047	2,047	512
15	15	10	12	4,095	2,047	1,365	2,047	2,047	1,024

(b)

Figure 7.17: Test patterns and simulation results of BISTSYN and previous techniques

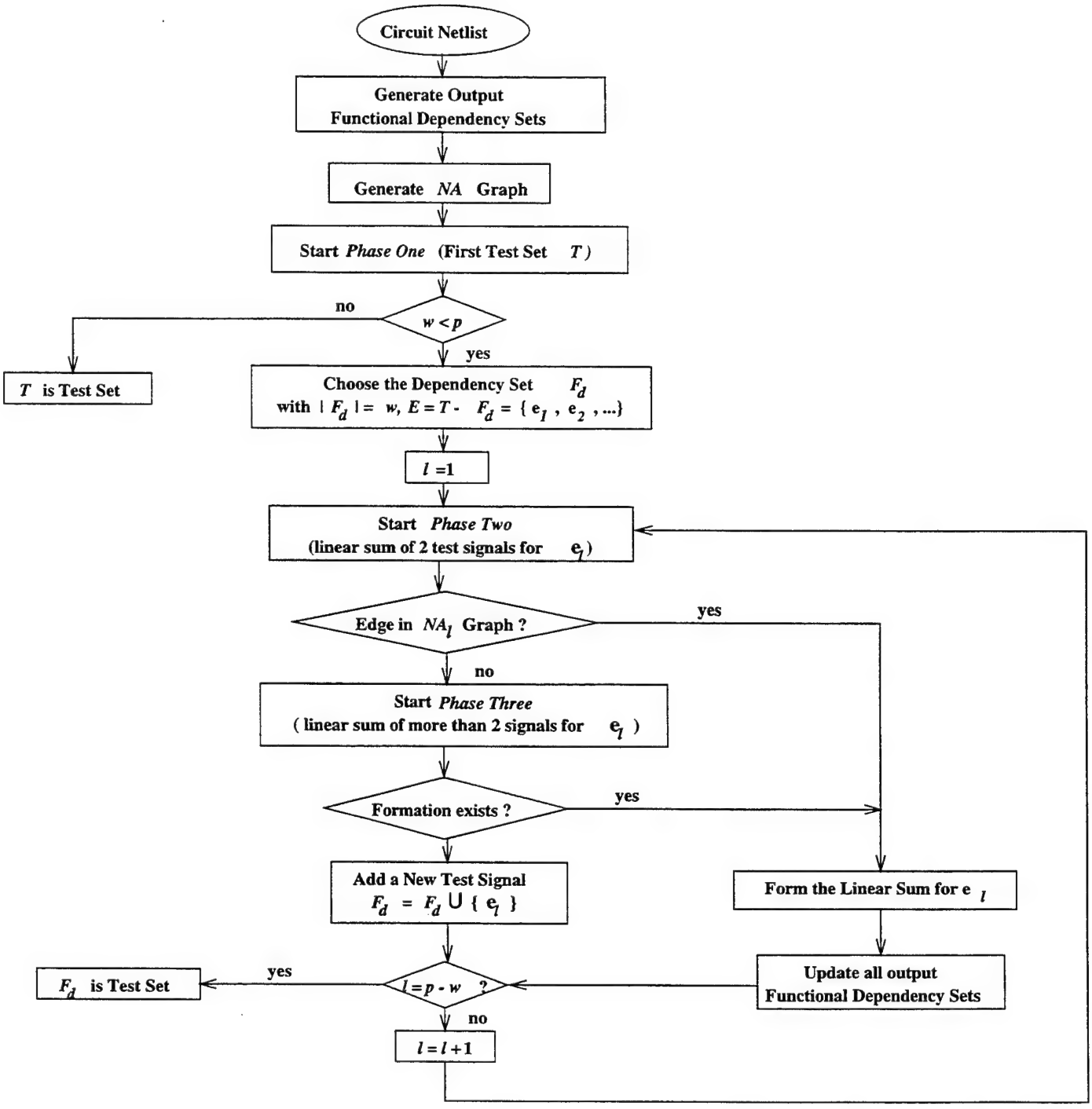


Figure 7.18: Flow chart of Three Phase Cluster Partitioning Algorithm

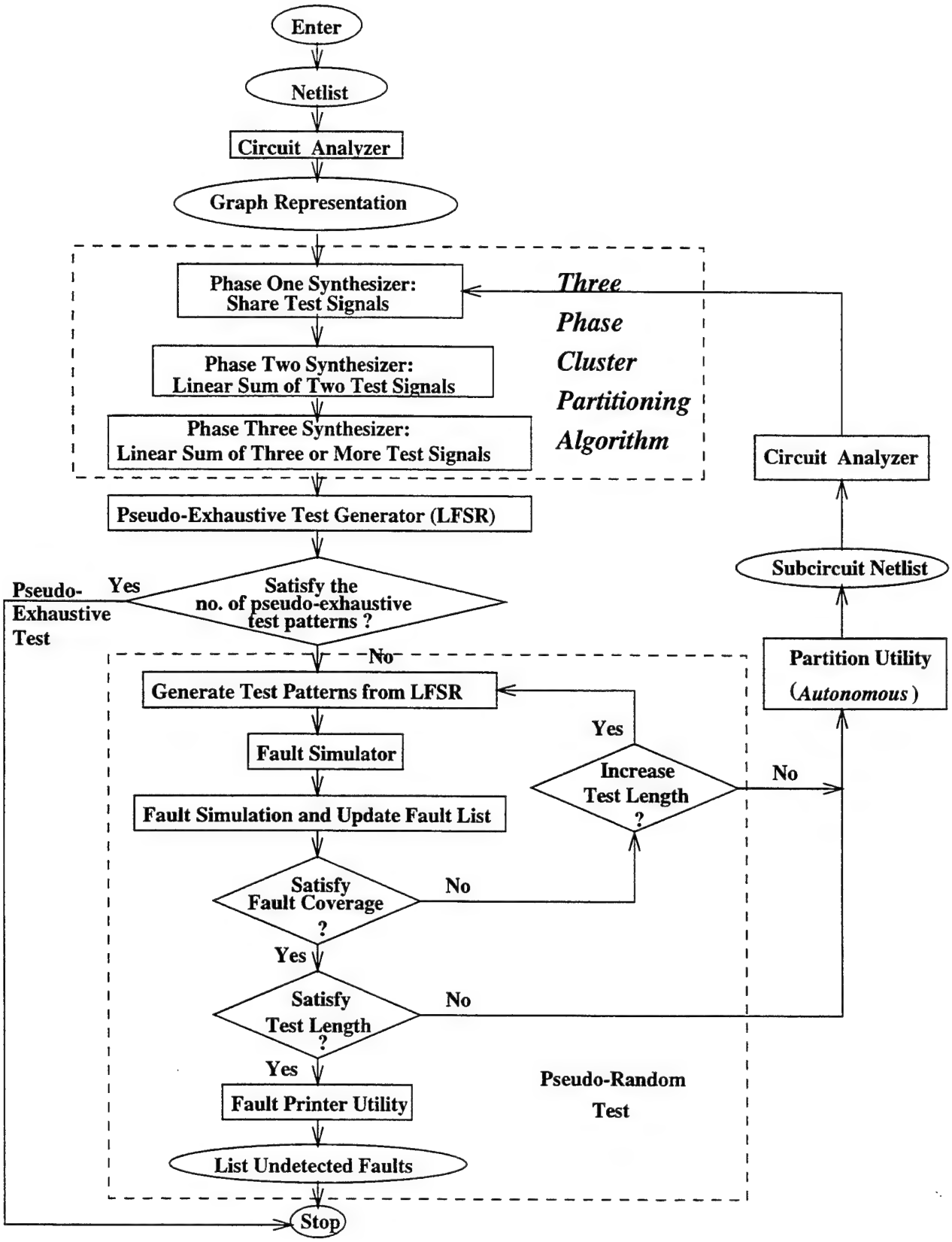


Figure 7.19: BISTSYN System

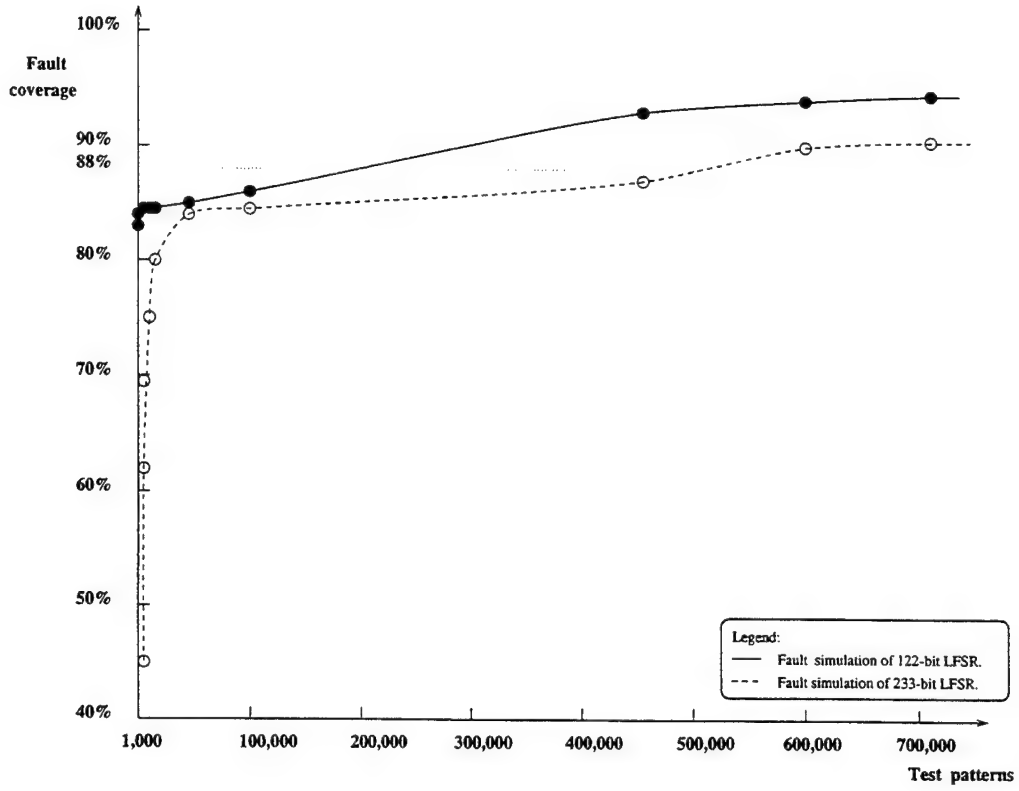


Figure 7.20: Fault simulations of C2670

length are both satisfied, the system will provide the user an option of printing the list of all the undetected faults and then exit. Otherwise, the system provides an option of partitioning the CUT by the partition utility, *Autonomous.* After the circuit is partitioned, each subcircuit will be analyzed by the circuit analyzer like the original CUT, and the same procedures are applied to each subcircuit.

7.5 Experimental Results

The design generator, *BISTSYN,* was developed to facilitate VLSI BIST designs with the procedure described in previous Sections. Written in C, the program is currently running on SUN 3 and Sparc Stations. Before we apply our programs to benchmark circuits and different VLSI designs, we use 42 random logics to evaluate "BISTSYN" and also compare the generated results with those produced by five previously proposed test generation methods ("SDC"[10], "LFSRs/XORs"[12], "CWC"[11], "Condensed LFSR"[14] and "LFSR with Cyclic Code"[15]). We list all the outputs along with the inputs that each output depends upon for these 42 random logic circuits in the Appendix. Figure 7.17(b) compares the number of required test patterns for all these 42 circuits. As seen from the Figure 7.17(b), "BISTSYN" requires fewer test patterns in almost all circuits except the one marked with "*" in which "CWC" requires 36 test patterns while "BISTSYN" needs 64 test patterns.

Circuit	STG3					ATPG without *Autonomous*					ATPG with *Autonomous*						
	Faults	test patterns	Untestable	Dropped	Fault Coverage	Faults	test patterns	Untestable	Dropped	Fault Coverage	Faults	test patterns	No. of subcircuits	Mux overhead	Untestable	Dropped	Fault Coverage
c432	524	75	1	3	99.24%	524	53	1	3	99.24%	584	82	3	13.46%	1	0	99.83%
c499	758	71	8	0	98.95%	758	55	8	0	98.945%	646	65	2	5.39%	8	0	98.76%
c880	942	106	0	0	100.00%	942	71	0	0	100.00%	1011	60	2	8.96%	0	0	100.00%
c1355	1574	145	8	0	99.49%	1574	90	8	0	99.49%	1526	99	2	4.36%	8	0	99.48%
c1908	1879	186	9	0	99.52%	1879	122	7	2	99.52%	2020	156	2	5.77%	2	0	99.90%
c2670	2747	214	103	14	95.74%	2747	114	86	31	95.74%	2969	158	2	15.56%	75	4	97.47%
c3540	3428	199	137	0	96.00%	3428	180	137	0	96.00%	3668	218	3	7.99%	94	0	97.44%
c5315	5350	308	59	0	98.88%	5350	138	59	0	98.88%	5492	205	3	9.40%	51	0	99.07%
c6288	7744	35	34	25	94.24%	7744	39	34	0	99.56%	8034	94	5	6.50%	32	2	99.58%
c7552	7550	398	131	1	98.25%	7550	232	71	0	99.05%	8018	274	2	11.57%	45	0	99.44%
Average	3250	174	49	4.3	98.03%	3250	109.4	41.1	3.6	98.64%	3397	141	2.6	8.90%	31.6	0.6	99.10%

Figure 7.21: Test generation results of benchmarks

7.5.1 Benchmark Circuits

Some experimental results of benchmark circuits obtained with our preliminary implementations are shown in Figures 7.21, 7.22 and 7.23. The results reported in this section is before data compaction. (The possibility of aliasing after compaction is assumed to be very small.) The data are given for running ATPG (employing the FAN algorithm [19] with parallel fault propagation), BISTSYN and Autonomous on benchmark examples, using Sun SparcStation CPU. The run time of *ATPG* and *BISTSYN* is on the order of seconds of CPU time. Figure 7.21 compares the number of required test patterns for the single stuck-at faults in these benchmarks. In Figure 7.21, the first column are results from "STG3"[20] which is a test generation system and is about one order of magnitude faster than GENTEST [21]. The second and third columns are ATPG results with and without *Autonomous* designated by "ATPG with Autonomous" and "ATPG without Autonomous" in Figure 7.21, respectively. As seen from the Figure 7.21, the number of untestable faults as well as the number of the dropped faults are greatly reduced for some benchmarks after they are partitioned. "ATPG with Autonomous" has the highest fault coverage in all the circuits.

For some partitioned subcircuits of benchmarks, the size of the pseudo-exhaustive test generator may be still relatively large even after reducing the number of required test signals by applying *Three-Phase Cluster Partitioning* algorithm.

For these types of subcircuits, the fault simulation based on the designed pseudo-exhaustive test generator is performed. Figure 7.22 compares the fault simulation results for these benchmarks. Figure 7.23 tabulates the ATPG and fault simulation of each subcircuit of benchmarks after they are partitioned by *Autonomous*. The fault coverage in ATPG is computed as the ratio of the number of testable faults to the total number of "equivalent" faults where [the number of testable faults] = [the number of equivalent faults] - [the number of untestable faults] - [the number of dropped faults]. Since some testable faults found in ATPG may not be detected by the fault simulation after a reasonable number of test

		Embedding BIST						*BISTSYN* without *Autonomous*						*BISTSYN* with *Autonomous*							
Circuit	PI	Faults	Counter size	Generated Tests	Undetected Faults	Fault Coverage	Fault Efficiency	Faults	LFSR size	Generated Tests	Undetected Faults	Fault Coverage	Fault Efficiency	Faults	Max LFSR size	No. of subcircuits	Mux overhead	Generated Tests	Undetected Faults	Fault Coverage	Fault Efficiency
c432	36	524	10	1024	10	98.09%	98.85%	524	36	4999	4	99.24%	100.00%	584	28	3	13.46%	6894	1	99.83%	100.00%
c499	41	758	10	1024	11	98.55%	99.60%	758	41	798	8	98.95%	100.00%	646	40	2	5.39%	862	8	98.76%	100.00%
c880	60	942	13	8192	0	100.00%	100.00%	942	45	15456	0	100.00%	100.00%	1011	23	2	8.96%	3795	0	100.00%	100.00%
c1355	41	1574	12	4096	8	99.49%	100.00%	1574	41	10000	8	99.49%	100.00%	1526	40	2	4.36%	2783	8	99.48%	100.00%
c1908	33	1879	13	8192	11	99.41%	99.90%	1879	33	4999	9	99.52%	100.00%	2020	28	2	5.77%	30000	2	99.90%	100.00%
c2670	233	2747	16	65536	398	85.51%	89.32%	2747	122	824113	149	94.57%	98.78%	2969	69	2	15.56%	623040	160	94.61%	97.20%
c3540	50	3428	13	8192	266	92.24%	96.08%	3428	50	99999	137	96.00%	100.00%	3668	31	3	7.99%	418780	94	97.44%	100.00%
c5315	178	5350	13	8192	80	98.50%	99.60%	5350	67	4999	59	98.88%	100.00%	5492	51	3	9.40%	7031	51	99.07%	100.00%
c6288	32	7744	9	512	83	98.93%	99.36%	7744	32	256	34	99.56%	100.00%	8034	27	5	6.50%	416	34	99.58%	100.00%
c7552	207	7550	N/A	N/A	N/A	N/A	N/A	7550	194	499999	202	97.33%	98.25%	8018	69	2	11.57%	16072	45	99.44%	100.00%
Average	91.1	3250	12.11	11662	96.3	96.75%	98.07%	3250	66	146562	61	98.35%	99.70%	3397	40.5	2.6	8.90%	110967	40.3	98.81%	99.72%

Figure 7.22: Fault simulation results of benchmarks

patterns are applied, the fault coverage in the fault simulation is computed as the ratio of the number of detected faults to the total number of "equivalent" faults where [the number of detected faults] = [the number of equivalent faults] - [the number of undetected faults]. The fault efficiency in fault simulation is computed as the ratio of the number of detected faults to the total number of testable faults. In Figure 7.22, the first column responds results from "Embedding BIST"[22]. The second and third columns are the fault simulation results with and without *Autonomous* by *BISTSYN* which are respectively designated by "BISTSYN without Autonomous" and "BISTSYN with Autonomous". Note that in the column of "BISTSYN with Autonomous", the number of test patterns is calculated as the sum of the number of test patterns required for testing each subcircuit. The fault coverage and the fault efficiency are calculated based on the sum of the total number of equivalent faults in each subcircuit. As shown in Figure 7.22, for all benchmark circuits, both "BISTSYN without Autonomous" and "BISTSYN with Autonomous" achieves very good fault coverage and fault efficiency results after a reasonable number of test patterns are generated from the designed pseudo-exhaustive test generator. The number of required test signals for pseudo-exhaustively testing these benchmark circuits after they are partitioned is significantly reduced in comparison with their primary inputs. This is indicated in the sub-column "max LFSR size" of Figure 7.22, which shows the maximum size of LFSRs used in self-testing the partitioned subcircuits. The sub-column "Mux overhead" indicates the overhead of multiplexers inserted between distinct subcircuits and it is calculated in terms of transistor pairs. As shown in Figure 7.22, the average overhead of multiplexers in the benchmarks is 8.90%.

7.6 Conclusions

A BIST synthesizer *BISTSYN* was proposed to design a test generator for pseudo-exhaustively testing a circuit in which none of the outputs depend on all of the inputs.

							ATPG			BISTSYN				
Circuits	No. of Subcircuits	Total Gates	PI	PO	Max. Dep.	Equivalent Faults	Test patterns	Untestable faults	Fault coverage	LFSR size	Generated tests	Undetected faults	Fault coverage	Fault efficiency
c432	1	158	36	50	28	430	33	0	100.00%	28	352	0	100.00%	100.00%
	2	48	30	10	26	69	20	0	100.00%	26	917	0	100.00%	100.00%
	3	47	27	4	27	85	29	1	98.82%	27	5,625	1	98.82%	100.00%
c499	1	121	41	40	14	274	10	0	100.00%	14	64	0	100.00%	100.00%
	2	162	40	32	40	372	55	8	97.85%	40	798	8	97.85%	100.00%
c880	1	264	60	59	17	572	33	0	100.00%	17	1,770	0	100.00%	100.00%
	2	224	44	11	23	439	60	0	100.00%	23	2,025	0	100.00%	100.00%
c1355	1	337	41	40	14	834	15	0	100.00%	14	51	0	100.00%	100.00%
	2	290	40	32	40	692	84	8	98.84 %	40	2,783	8	98.84 %	100.00%
c1908	1	656	33	116	14	1,211	21	0	100.00%	15	50	0	100.00%	100.00%
	2	241	116	25	28	807	175	2	99.75%	28	2,048	2	99.75%	100.00%
c2670	1	791	233	254	14	1,582	45	23	98.55%	14	256	23	98.55%	100.00%
	2	771	136	22	69	1,387	117	56	95.96%	69	622,784	137	90.12%	93.91%
c3540	1	1,204	50	129	27	2,477	133	48	98.06%	28	16,384	48	98.06%	100.00%
	2	493	116	34	31	850	53	16	98.12%	31	2,048	16	98.12%	100.00%
	3	174	36	11	23	341	30	30	91.20%	31	400,000	30	91.20%	100.00%
c5315	1	2,265	178	299	47	4,902	115	51	98.96%	47	4,300	51	98.96%	100.00%
	2	365	170	20	51	498	64	0	100.00%	51	1,995	0	100.00%	100.00%
	3	55	30	4	12	92	26	0	100.00%	12	736	0	100.00%	100.00%
c6288	1	1,572	32	137	20	5,014	36	32	99.36%	21	128	32	99.36%	100.00%
	2	462	82	54	20	1,304	17	0	100.00%	20	96	0	100.00%	100.00%
	3	481	86	58	23	1,343	19	0	100.00%	23	96	0	100.00%	100.00%
	4	103	27	26	27	272	12	2	99.27%	27	64	2	99.27%	100.00%
	5	39	13	12	11	101	10	0	100.00%	11	32	0	100.00%	100.00%
c7552	1	3,719	207	108	53	7,550	232	44	99.27%	73	7,072	44	99.27%	100.00%
	2	1,305	469	52	69	1,973	159	1	99.95%	69	10,000	1	99.95%	100.00%

Figure 7.23: ATPG and fault simulation results after partitioning

The procedure consists of three phases. Using only phase one, a minimum number of test signals required for pseudo-exhaustively testing a CUT without linear sum is found. For a CUT with the non-MTC property, phase two and phase three procedures are subsequently applied to further reduce the number of test signals that are required for pseudo-exhaustive test. The hierarchical design procedure minimizes the number of test patterns that are required for pseudo-exhaustive test. The experimental results are very favorably compared with previous techniques. Moreover, it produces test generation circuitry with low hardware overhead.

For those conventional circuits which are extremely unsuitable for pseudo-exhaustive test, a partitioning tool, *Autonomous,* is employed to partition a circuit into subcircuits so that each subcircuit is more suitable for pseudo-exhaustive test. Several benchmarks are

used to evaluate the circuit partitioning. The average number of untestable faults as well as the average number of dropped faults after circuit partitioning is 27.96% less than that before circuit partitioning and is 39.59% less than that of "STG3" method. When used with *Autonomous* on a set of benchmarks, *BISTSYN* leads to a BIST design achieving a higher fault coverage compared to previous work. The experimental results show that the average fault coverage achieved by the *BISTSYN* is 98.35% and 98.81% before and after circuit partitioning respectively while the fault coverage achieved by the "Embedding BIST" method is 96.75%. The average fault efficiency achieved by *BISTSYN* is 99.70% and 99.72% before and after circuit partitioning respectively while the average fault efficiency achieved by "Embedding BIST" is 98.07%.

Appendix

PI PO W	Dependency
5 5 3	F1(1,2,3)
	F2(1,3)
	F3(2,4,5)
	F4(1,2,3)
	F5(1,5)
5 5 4	F1(1,2)
	F2(1,3)
	F3(1,2,5)
	F4((1,2,3,4)
	F5(1,2,5)
5 5 5	F1(1,2,3,4)
	F2(1,2,3,4,5)
	F3(1,2)
	F4(3,5)
	F5(1,2,3,4,5)
8 8 2	F1(2,8)
	F2(7,8)
	F3(1,2)
	F4(3,4)
	F5(1,2)
	F6(2,8)
	F7(4,7)
	F8(1,6)
8 8 3	F1(2,7,8)
	F2(4,7)
	F3(1,2,8)
	F4(1,2,7)
	F5(5,8)
	F6(4,7)
	F7(1,4,6)
	F8(1,2,3)
8 8 4	F1(2,8)
	F2(7,8)
	F3(1,2,8)
	F4(1,2,3,7)
	F5(2,5,8)
	F6(4,5,6,7)
	F7(3,6)
	F8(1,2)
8 8 5	F1(2,4,7,8)
	F2(1,2,4,5,8)
	F3(1,3)
	F4(2,3)
	F5(2,3,4,5,7)
	F6(1,6)
	F7(1,2,3,4)
	F8(4,8)
8 8 6	F1(2,7,8)
	F2(2,4,5,7)
	F3(1,3,4,8)
	F4(2,3)
	F5(2,3,4,7)
	F6(4,6)
	F7(1,2,3,4,6,7)
	F8(4,8)
8 8 7	F1(1,2,3,4,7,8)
	F2(1,2,3,4,5,6,8)
	F3(2,3,4,5,8)
	F4(4,5,6,7,8)
	F5(1,2,3,4,5,6,7)
	F6(1,3,4,8)
	F7(1,2,3,4,5,6,8)
	F8(1,2,4,5,6,8)
10 10 2	F1(2,6)
	F2(1,8)
	F3(7,10)
	F4(1,10)

PI PO W	Dependency
10 10 4	F3(1,7,10)
	F4(1,2,3,7)
	F5(2,5,6)
	F6(1,4,5,9)
	F7(7,10)
	F8(1,5)
	F9(3,4,8,10)
	F10(3,6)
10 10 5	F1(1,2,3,6)
	F2(1,2,7,8,10)
	F3(1,2)
	F4(3,10)
	F5(2,4,5,6,9)
	F6(8,9)
	F7(1,5,6,9)
	F8(4,10)
	F9(4,10)
	F10(6,7,8)
10 10 6	F1(1,2,6)
	F2(1,2,8,10)
	F3(1,2,4,10)
	F4(3,10)
	F5(2,4,6,9)
	F6(4,8)
	F7(1,2,5,6,9,10)
	F8(8,10)
	F9(3,6,7,8,9,10)
	F10(2,3)
10 10 7	F1(1,2 3,4,6,8)
	F2(1,2,3,4,5,7,10)
	F3(2,3,5,6,10)
	F4(1,4,5,8,9)
	F5(1,2,3,5,6,9,10)
	F6(3,4,8,10)
	F7(1,2,3,4,6,7,8)
	F8(1,2,3,4,5,10)
	F9(2,3,5)
	F10(2,3,4,5,6,7,8)
10 10 9	F1(1,2,6)
	F2(1,8)
	F3(1,2,3,4,5,6,7,8,10)
	F4(2,4,5,6,7,9)
	F5(8,9)
	F6(1,2,4,5,6,9,10)
	F7(3,4)
	F8(1,2,3,4,6,7,8,9,10)
	F9(2,4,10)
	F10(1,2,3,4,5,6,10)
10 10 10	F1(1,2,3,4,5,6,8,9,10)
	F2(1,2,3,4,5,6,7,8,10)
	F3(1,2,4,5,6,7,8,9,10)
	F4(1,5,9,10)
	F5(1,3,4,8,10)
	F6(1,2,3,4,6,7,8,9,10)
	F7(1,2,3,4,5,6,10)
	F8(1,2,3,4,5,6,7,8,9,10)
	F9(1,9)
	F10(2,10)
10 15 2	F1(2,6)
	F2(1,8)
	F3(7,10)
	F4(1,10)
	F5(3,10)
	F6(2,6)
	F7(4,9)
	F8(8,9)
	F9(6,9)

PI PO W	Dependency
10 15 3	F13(1,10)
	F14(1,4,10)
	F15(2,5)
10 15 4	F1(2,6)
	F2(1,8)
	F3(1,7,10)
	F4(1,2,3,7)
	F5(2,5,6)
	F6(1,4,5,9)
	F7(7,10)
	F8(1,5)
	F9(3,4,8,10)
	F10(3,6)
	F11(1,7)
	F12(2,7)
	F13(2,10)
	F14(2,10)
	F15(2,3,4,10)
10 15 5	F1(1,2,3,6)
	F2(1,2,7,8,10)
	F3(1,2)
	F4(3,10)
	F5(2,4,5,6,9)
	F6(8,9)
	F7(1,5,6,9)
	F8(4,10)
	F9(4,10)
	F10(6,7,8)
	F11(2,3)
	F12(1,2,4,10)
	F13(1,2,5)
	F14(3,10)
	F15(2,3,4,5,8)
10 15 6	F1(1,2,6)
	F2(1,2,8,10)
	F3(1,2,4,10)
	F4(3,10)
	F5(2,4,6,9)
	F6(4,8)
	F7(1,2,5,6,9,10)
	F8(8,10)
	F9(3,6,7,8.9,10)
	F10(2,3)
	F11(1,2,4,10)
	F12(1,2,3,4,5,10)
	F13(2,3,4,5,6,8)
	F14(1,2,3,4,9,10)
	F15(1,3,4,9,10)
10 15 7	F1(1,2,3,4,6,8)
	F2(1,2,3,4,5,7,10)
	F3(2,3,5,6,10)
	F4(1,4,5,8,9)
	F5(1,2,3,5,6,9,10)
	F6(3,4,8,10)
	F7(1,2,3,4,6,7,8)
	F8(1,2,3,4,5,10)
	F9(2,3,5)
	F10(2,3,4,5,6,7,8)
	F11(1,2,3,4,5,9,10)
	F12(1,2,3,4,9)
	F13(3,4,5,6,7,8)
	F14(5,6,9,10)
	F15(3,7,8)
10 15 8	F1(1,2,3,6,8)
	F2(1,10)
	F3(1,2,4,10)
	F4(3,5,10)

PI PO W	Dependency
10 15 9	F8(1,2,3,4,6,7,8,9,10)
	F9(2,4,10)
	F10(1,2,3,4,5,6,10)
	F11(1,2,3,4,5,6,7,8,9)
	F12(1,2,3,4,5,6,9,10)
	F13(2,10)
	F14(3,6,7,8)
	F15(5,6,8,9,10)
15 10 2	F1(1,2)
	F2(8,11)
	F3(7,15)
	F4(5,11)
	F5(3,5)
	F6(2,11)
	F7(9,14)
	F8(8,14)
	F9(6,14)
	F10(9,15)
15 10 3	F1(1,2,3)
	F2(11,13)
F3(7,14,15)	
	F4(3,6,12)
	F5(5,11)
	F6(6,9)
	F7(4,8,14)
	F8(6,11,14)
	F9(4,15)
	F10(3,14)
15 10 4	F1(1,2)
	F2(8,11)
	F3(1,7,15)
	F4(3,6,11,12)
	F5(2,5,11)
	F6(4,6,9,14)
	F7(7,10)
	F8(5,11)
	F9(3,4,13,15)
	F10(11,13)
15 10 5	F1(1,2,3,4)
	F2(1,2,7,13,15)
	F3(6,11)
	F4(5,8)
	F5(2,6,9,14,15)
	F6(8,14)
	F7(5,6,11,14)
	F8(4,15)
	F9(10,14)
	F10(6,7,8)
15 10 6	F1(1,2,3)
	F2(1,11,13,15)
	F3(5,6,11,14)
	F4(5,8)
	F5(2,6,9,14)
	F6(4,8)
	F7(5,6,9,10,11,14)
	F8(13,15)
	F9(6,7,8,10,11,13)
	F10(2,7)
15 10 7	F1(1,2,3,4,8,11)
	F2(1,5,6,7,11,14,15)
	F3(2,5,6,8,11)
	F4(4,6,8,9,14)
	F5(5,6,9,10,11,14,15)
	F6(3,10,13,14)
	F7(2,3,6,7,8,9,10)
	F8(1,4,5,7,10,15)
	F9(3,7,15)

F5(3,10)
F6(2,6)
F7(4,9)
F8(8,9)
F9(6,9)
F10(9,10)

10 10 3 F1(1,2,6)
F2(1,8)
F3(4,7,10)
F4(1,3,7)
F5(5,6)
F6(6,9)
F7(4,8,9)
F8(1,6,9)
F9(4,10)
F10(3,4)

10 10 4 F1(2,6)
F2(1,8)

15 10 10 F8(3,5,6,7,8,9,
10,11,14,15)
F9(6,14)
F10(2,15)

15 15 2 F1(1,2)
F2(8,11)
F3(7,15)
F4(5,11)
F5(3,5)
F6(2,11)
F7(9,14)
F8(8,14)
F9(6,14)
F10(9,15)
F11(3,13)
F12(11,13)
F13(6,7)
F14(2,7)
F15(5,7)

15 15 3 F1(1,2,3)
F2(11,13)
F3(7,14,15)
F4(3,6,12)
F5(5,11)
F6(6,9)
F7(4,8,14)
F8(6,11,14)
F9(4,15)
F10(3,14)
F11(6,7,11)
F12(2,3)
F13(1,15)
F14(4,10,15)
F15(7,15)

15 15 4 F1(1,2), F2(8,11)

F10(9,10)
F11(3,8)
F12(3,6)
F13(1,7)
F14(2,7)
F15(2,10)

10 15 3 F1(1,2,6)
F2(1,8)
F3(4,7,10)
F4(1,3,7)
F5(5,6)
F6(6,9)
F7(4,8,9)
F8(1,6,9)
F9(4,10)
F10(3,4)
F11(6,7,8)
F12(2,3)

15 15 4 F3(1,7,15)
F4(3,6,11,12)
F5(2,5,11)
F6(4,6,9,14)
F7(7,10)
F8(5,11)
F(3,4,13,15)
F10(11,13)
F11(6,7)
F12(2,7)
F13(5,7)
F14(2,10)
F15(3,5,7,8)

15 15 5 F1(1,2,3,4)
F2(1,2,7,13,15)
F3(6,11)
F4(5,8)
F5(2,6,9,14,15)
F6(8,14)
F7(5,6,11,14)
F8(4,15)
F9(10,14)
F10(6,7,8)
F11(2,7)
F12(4,5,7,15)
F13(1,2,15)
F14(5,8)
F15(3,5,7,8,14)

15 15 6 F1(1,2,3)
F2(1,11,13,15)
F3(5,6,11,14)
F4(5,8)
F5(2,6,9.14)
F6(4,8)
F7(5,6,9,10,11,14)

F5(1,4,5,6,9)
F6(7,9)
F7(1,2,5,9,10)
F8(3,4,8,10)
F9((1,2,3,6,7,8)
F10(1,2,3,10)
F11(1,2,3,4,5,6,10)
F12(2,5)
F13(2,3,4,5,9)
F14(1,2,3,4,5,9,10)
F15(2,3,4,5,6,8,9,10)

10 15 9 F1(1,2,6)
F2(1,8)
F3(1,2,3,4,5,6,7,8,10)
F4(2,4,5,6,7,9)
F5(8,9)
F6(1,2,4,5,6,9,10)
F7(3,4)

15 15 6 F8(13,15)
F9(6,7,8,10,11,13)
F10(2,7)
F11(4,5,7,15)
F12(1,2,3,5,7,15)
F13(3,5,7,8,14,15)
F14(1,6,8,11,14,15)
F15(4,8,9,11,14)

15 15 7 F1(1,2,3,4,8,11)
F2(1,5,6,7,11,14,15)
F3(2,5,6,8,11)
F4(4,6,8,9,14)
F5(5,6,9,10,11,14,15)
F6(3,10,13,14)
F7(2,3,6,7,8,9,10)
F8(1,4,5,7,10,15)
F9(3,7,15)
F10(3,5,7,8,10,14,15)
F11(1,2,6,8,11,14,15)
F12(4,7,8,9,11)
F13(1,2,3,5,8,14)
F14(1,5,9,10)
F15(12,13,14)

15 15 8 F1(1,2,3,4,8)
F2(1,15)
F3(5,6,11,14)
F4(5,6,8)
F5(6,7,9,14,15)
F6(7,14)
F7(5,9,11,14,15)
F8(3,10,13,14)
F9(2,3,6,7,8,9)
F10(1,5,7,15)
F11(1,2,3,5,7,10,15)

F10(3,5,7,8,10,14,15)

15 10 9 F1(1,2,3)
F2(11,13)
F3(3,5,6,7,8,11,12,14,15
F4(2,6,9,11,14,15)
F5(8,14)
F6(4,5,6,9,11,14,15)
F7(3,14)
F8(2,3,4,6,7,8,9,10,11)
F9(4,5,7)
F10(1,2,3,5,7,8,15)

15 10 10 F1(1,2,3,4,5,8,11,13,15)
F2(3,5,6,7,8,9,11,12,14)
F3(1,4,6,7,8,9,10,14,15)
F4(5,9,11,14)
F5(3,10,13,14,15)
F6(2,3,4,6,7,8,9,10,15)
F7(1,2,3,4,5,10,15)

15 15 8 F12(7,15)
F13(5,7,8,9,14)
F14(1,2,8,9,11,14,15)
F15(3,4,5,6,7,8,12,14)

15 15 9 F1(1,2,3)
F2(11,13)
F3(3,5,6,7,8,11,12,14,15
F4(2,6,9,11,14,15)
F5(8,14)
F6(4,5,6,9,11,14,15)
F7(3,14)
F8(2,3,4,6,7,8,9,10,11)

15 15 9 F9(4,5,7)
F10(1,2,3,5,7,8,15)
F11(3,5,6,7,8,9,10,14,15
F12(1,2,8,9,10,11,14,15)
F13(5,12)
F14(1,2,3,8)
F15(1,5,8,9,10)

15 15 10 F1(1,2,3,4,5,8,11,13,15)
F2(3,5,6,7,8,9,11,12,14)
F3(1,4,6,7,8,9,10,14,15)
F4(5,9,11,14)
F5(3,10,13,14,15)
F6(2,3,4,6,7,8,9,10,15)
F7(1,2,3,4,5,10,15)
F8(3,5,6,7,8,9,10,11,14,
F9(6,14)
F10(2,15)
F11(8,9)
F12(5,7,12)
F13(1,2,3,4,8,9,15)
F14(1,5,6,8,9,10,11,12,1
F15(2,3,5,6,7,10,15)

Chapter 8

Circuit Partitioning Tool: AUTONOMOUS

8.1 Introduction

While developing the next-generation digital VLSI products, we recognize the need for an expanded testability and maintainability strategy. These new products will contain as many as 1000,000 logic gates operating at clock-rate of 100 MHz. Poor testability in circuits of this complexity can have devastating effects on the cost of testing and maintaining the boards and systems which use them.

Built-In Self-Test (BIST) has been proposed as a powerful solution to the VLSI testing problem. Pseudo-exhaustive test is a BIST design methodology that provides effective, 100-percent fault coverage for all testable stuck-at faults. However, the pseudo-exhaustive test is not suitable to "total dependency" circuits in which at least one output is functionally dependent on all the inputs. Even for a "partial dependency" circuit, the size of a pseudo-exhaustive test set may be still too large to be applicable in practice. In such a case, pseudo-exhaustive test can be achieved by partitioning techniques. The principle is to partition the circuit into subcircuits such that the output of each subcircuit is functionally dependent on only a small number of its inputs. Then the subcircuits can be pseudo-exhaustively tested. Since the time complexity of test generation and fault simulation grows faster than a linear function of circuit size, it is cost-effective to partition large circuits to reduce these costs. A circuit partitioning method is presented to partition the digital combinational portions of a circuit into different structural subcircuits so that each subcircuit can be pseudo-exhaustively tested. Furthermore, a tight lower bound on the number of interconnections between distinct subcircuits is derived. These interconnections are required for self-testing each subcircuit when the circuit is partitioned into a specified number of groups of specified sizes. We demonstrate the effectiveness of the partitioning method by illustrations of applying the method to circuit examples and benchmark circuits.

For those conventional circuits which are extremely unsuitable for pseudo-exhaustive test (i.e., there exists at least one output of such a circuit that is functionally depen-

dent on all the inputs), circuit partitioning is a method to limit the number of inputs driving each subcircuit and therefore makes the pseudo-exhaustive test possible. Several heuristic algorithms have been proposed for solving the circuit partitioning problem [40, 41, 42, 43, 44, 45, 46, 47, 48]. In [40], a partitioning scheme with multiplexers to gain access to segments is first proposed. But no partitioning solution for large scale circuit is discussed. In [41], a linear time algorithm was proposed to partition the tree circuits (fanout free circuits). However, the technique can not be applied to most practical circuits which have reconvergent fanout loops. In [42], an iterative algorithm for circuit segmentation is proposed to generate the dependency cones moving from one segment to another. In [43], a method was proposed to calculate the smallest number of test points added into the fanout free circuit. For circuits with fanouts, determining the smallest number of test points for the circuit becomes NP-complete [44], that is, there is no computationally practical way to obtain the smallest set of test points. An interesting algorithm based on simulated annealing is proposed in [45] to segment a digital combinational circuit. The feasibility of the approach is demonstrated through some circuits of size up to 3,500 gates. Other methods of segmenting a circuit by adding the bypass storage cells for pseudo-exhaustive test were presented in [46, 47, 48]. These storage cells added to the circuit will induce extra hardware and delay overhead.

Exhaustive testing of a combinational circuit involves exercising the circuit with all possible input patterns. It ensures detection of all testable combinational faults in the circuit. A combinational fault does not manifest any sequential behavior of the circuit and is testable with a single input pattern. However, for circuits with large number of inputs, exhaustive testing is very time consuming and may not be practical. To reduce the test time without compromising on the coverage of the stuck-at faults, a circuit can be tested pseudo-exhaustively. Pseudo-exhaustive testing ensures detection of all testable combinational faults in the circuit. In pseudo-exhaustive testing, the circuit is partitioned into subcircuits and each subcircuit is tested exhaustively. The time required for pseudo-exhaustive testing depends on the sizes of the subcircuits. Although the pseudo-exhaustive testing is very promising, it is very difficult to find a good partitioning algorithm. If a circuit partitioning algorithm is not efficient, the number of interconnections among the subcircuits may be very large. Since each interconnection will induce extra hardware to physically separate the subcircuits in test mode, the hardware overhead may not be tolerable.

A circuit partitioning tool, *Autonomous*, is presented in this chapter to employ a circuit partitioning scheme such that the combinational portions of the circuit are partitioned into different structural subcircuits and therefore, each of which can be pseudo-exhaustively tested. In fact, it is necessary to have each partitioned subcircuit with the property that each output is functionally dependent on sufficiently small number of inputs. Access to the embedded inputs and outputs of each subcircuit can be achieved by inserting multiplexers (Mux's) and connecting the embedded inputs and outputs of each subcircuits to those primary inputs and outputs that are not used by the subcircuits.

We have formulated the partitioning problem as the classical integer linear programming problem. For small circuits, optimal solutions can be obtained by solving the integer

linear programming formulation. However, the formulation may not be computationally viable for very large circuits since the number of constraints grows non-linearly with the number of levels in the circuit. We have developed an efficient heuristic partitioning procedure applicable to large circuits. The results of combinational benchmark circuits and comparison with the results of others validates the efficiency of our heuristic approach.

The chapter is organized as follows. Section 8.2 presents the partitioning strategy and the problem formulation. The heuristic procedure for computing the lower bound of optimal partition is described in Section 8.3.

8.2 Circuit Partitioning

Consider a combinational circuit with n inputs and m outputs. An output is said to depend on an input if there exists at least one path from that input to the output. The set of inputs on which an output depends is referred to as the dependency set of the output. The dependency value for an output is given by the cardinality of its dependency set. Assume each output has a dependency value of i or less and at least one output has the dependency of exact i inputs. The circuit can be characterized as an (n, m, i) circuit. Exhaustive testing of the circuit requires 2^n test patterns that may be prohibitive for large values of n.

For pseudo-exhaustive testing, we shall consider partitioning that forms the partitioned subcircuits. The (n, m, i) circuit can be partitioned into disjointed subcircuits and each subcircuit can be exhaustively tested with 2^i test patterns. If i is sufficiently smaller than n, the test time taken for pseudo-exhaustive testing can be much less compared to exhaustive testing.

For some circuit, i can be large enough to prohibit the testing even with the least required 2^i patterns. To reduce the test time, the circuit can be partitioned into two or more subcircuits such that the dependency value for each output of the partitioned subcircuits can be restricted to some predetermined value, l where $l < i$. This is referred to as circuit partitioning problem and can be achieved by partitioning the circuit by placing few Mux's (say s). In the test mode, the original (n, m, i) circuit is modified to an $(n + s, m + s, l)$ circuit. For example, consider the $(8, 2, 8)$ circuit shown in Figure 8.1. The circuit requires $2^8 = 256$ patterns for both exhaustive and pseudo-exhaustive testing. The circuit can be partitioned into two subcircuits by placing 2 Mux's such that the dependency value for each output of the subcircuits is no more than 4. In the test mode, the circuit becomes an $(10, 4, 4)$ circuit and requires $2^4 + 2^2 = 20$ patterns for pseudo-exhaustive testing.

Figure 8.2 is an example circuit from [46] used for the illustration of the logic partitioning to pseudo-exhaustive test. The circuit is a "total dependency" $(13, 2, 13)$ circuit. Without the logic partition, 13 test signals ($2^{13} = 8,192$ test patterns) are required for exhaustively testing the circuit. While using Autonomous, the circuit is partitioned into two subcircuits C_1 and C_2 and becomes an $(17, 6, 6)$ circuit shown in Figure 8.2 where C_1 has 11 inputs and 4 outputs while C_2 has 6 inputs and 2 outputs. By mapping the

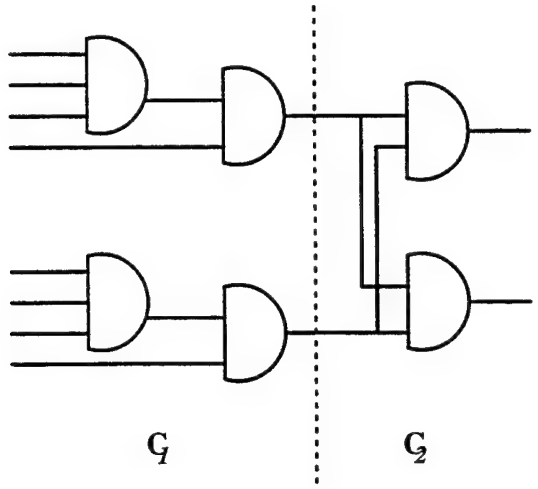


Figure 8.1: Example of an (8 2 8) circuit

partitioned circuit to Figure 7.2(a), A represents the signals to PI 3-13; C represents the signals to PI 1 and PI 2; $B = E = F = \phi$; D is the set of nodes: 64, 67, 70, 73 which are depicted in Figure 8.2. G represents the signals from PO's. The maximum dependency of C_1 is $w_1 = 4$ and that of C_2 is $w_2 = 6$. It is found that 4 test signals (2^4 test patterns) are enough for pseudo-exhaustively testing C_1. During the testing, PI 1, 2, 6, 10 and 12 share a test signal, and PI 3, 7 and 13 share a test signal, and PI 4, 8 and 11 share a test signal, and PI 5 and 9 share a test signal. The maximum dependency of C_2 is $w_2 = 6$. It is found that 6 test signals (2^6 test patterns) can pseudo-exhaustively testing C_2. So $2^4 + 2^6 = 80$ test patterns are enough to pseudo-exhaustively test the circuit, which is much less than 8,192 test patterns required by exhaustive test and is also favorably compared with 96 test patterns required by the pseudo-exhaustive technique [46] where the 96 test patterns need to be stored in the on-chip memory for self-testing.

Our objective is to partition the circuit by placing Mux's in order to reduce the dependency values of the circuit to an acceptable number. Mux's add area overhead to the original circuit. Hence, it is desirable to have a minimum number of these cells to achieve our objective.

The following graph model is used to represent a circuit. A combinational circuit C is represented by an undirected graph G which has nodes $N = \{1, ..., n\}$ and edge set E. The nodes are corresponding to PI's or PO's or gates of C. An undirected edge of G represents a signal flow between the corresponding components of C. For a large scale circuit, our interest is to partition the nodes of G into disjoint subsets S_1, S_2,...., S_k, $k \geq 2$, in the following way:

1. The maximum output dependency value w_i for each sub-graph G_i, $i = 1, ..., k$, is less than or equal to the user specified size l of linear feedback shift register (LFSR). In the other words, the number of reduced exhaustive test patterns of each subcircuit is not great than 2^l

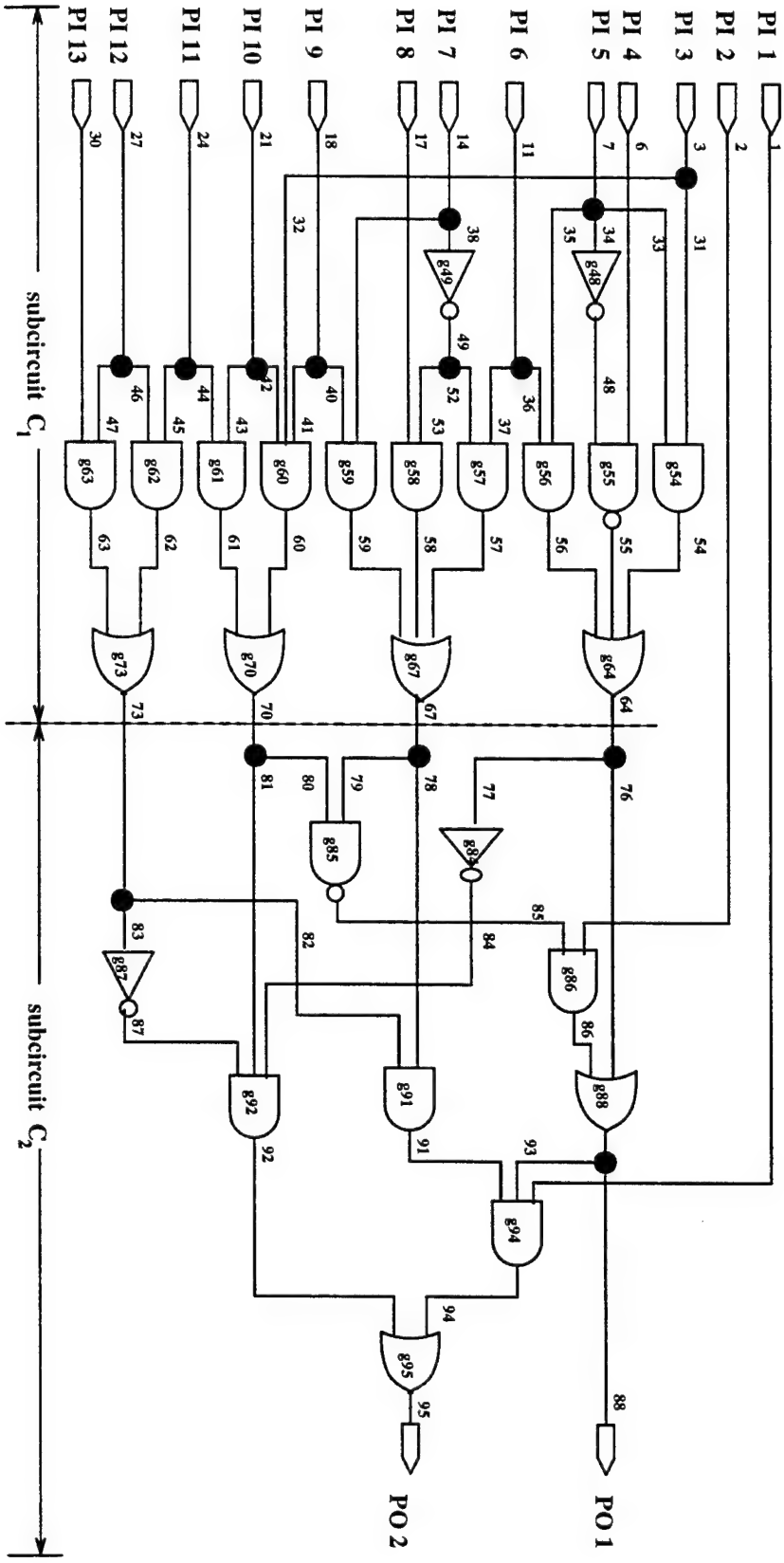


Figure 8.2: Example circuit

2. The number of edges joining nodes in distinct sub-graphs of the partition is minimized, i.e., the number of interconnections between distinct subcircuits is minimized. In the other words, the number of Mux's added between distinct subcircuits is minimized.

Several heuristic algorithms have been proposed for solving graph partitioning problems. In this chapter, we'll explore the graph partitioning techniquesSeveral heuristic algorithms have been proposed for solving graph partitioning problems. In this chapter, we'll explore the graph partitioning techniques [49, 50, 51] to achieve the above partitioning objectives.

Let $N = S_1 \cup S_2 \cup ... \cup S_k$ be a partition of the nodes of G. Let Let

$$X_j = (x_{1j}, x_{2j}, ...x_{nj})^T$$

be an indicator vector for S_j, j = 1,2, ..., k. Thus,

$$x_{ij} = \begin{cases} 1 & if\ i \in S_j, \\ 0 & if\ i \notin S_j. \end{cases}$$

Clearly,

$$\sum_{i=1}^{n} x_{ij} = \mid S_j \mid = m_j \quad for\ j\ = 1, 2, ..., k.$$

where m_j is the number of nodes in the j^{th} subcircuit. And,

$$\sum_{j=1}^{k} x_{ij} = 1 \quad for\ i\ =\ 1, 2, ..., n.$$

The number of edges with both endpoints in S_j is given by in S_2 is given by

$$\frac{1}{2} \sum_{r \in S_j} \sum_{s \in S_j} a_{rs} = \frac{1}{2} \sum_{r=1}^{n} \sum_{s=1}^{n} a_{rs} \times x_{rj} \times x_{sj} = \frac{1}{2} {X_j}^T \times \mathbf{A} \times X_j$$

Let a_{ij} denote the number of edges connecting node i and j and let $\mathbf{A} = (a_{ij})$ denote the adjacency matrix for G. Thus, the number of edges not cut by the circuit partitioning is denoted as E_{nc} and calculated in the following:

$$E_{nc} = \frac{1}{2} \sum_{j=1}^{k} {X_j}^T \times \mathbf{A} \times X_j$$

Let E_c denote the number of edges cut. We know $E_c + E_{nc} = \mid E \mid$. The problem can be reformulated as the following.

$$Maximize \sum_{j=1}^{k} {X_j}^T \times \mathbf{A} \times X_j$$

Subject to

$$\sum_{i=1}^{n} x_{ij} = m_j,\ where\ j\ = 1, 2, ..., k.$$

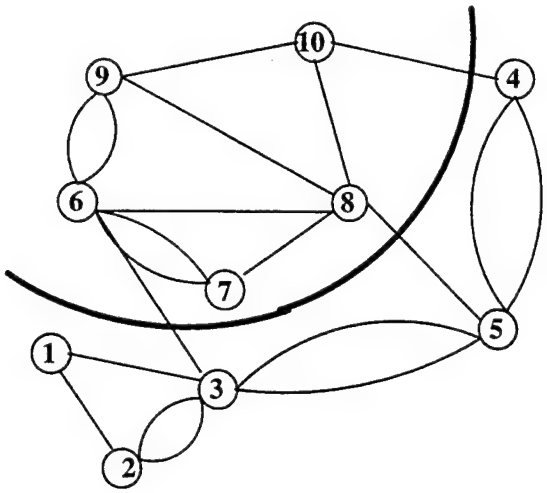


Figure 8.3: Example 1: graph partitioning

$$\sum_{j=1}^{k} x_{ij} = 1, \ where \ i \ = \ 1,2,...,n. \tag{8.1}$$

$$x_{ij} \ = \ 0 \ or \ 1 \ for \ all \ i \ and \ j.$$

$$\sum_{j=1}^{k} m_j = n$$

$$w_1, w_2, ..., w_k \leq l$$

Example 1: Consider the problem of partitioning the nodes of the graph G of Figure 8.3 into two sets containing 5 nodes each. The partition $S_1 \cup S_2 = \{1,2,3,4,5\} \cup \{6,7,8,9,10\}$ cuts 3 edges and appears to be optimal. For circuit partition, that means nodes 1,2,3,4,5 are allocated to subcircuit S_1 and nodes 6,7,8,9,10 are allocated to subcircuit S_2. The 3 cutting edges represent the number of interconnections between subcircuits S_1 and S_2.

Let $\mathbf{A} = (a_{ij})$ denote the adjacency matrix for G of example 1. Then,

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 2 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 2 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Applying the integer programming analysis to this partition example, the results show that $X_1 = (\ 1,\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0\ )^T$ and $X_2 = (\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 1,\ 1\ )^T$.

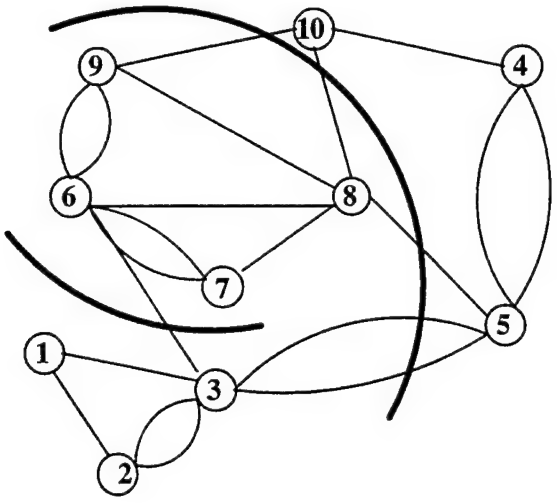


Figure 8.4: Example 2: graph partitioning

Thus, the number of edges not cut by the circuit partitioning is denoted as E_{nc} and calculated in the following:

$$E_{nc} = \frac{1}{2}\sum_{j=1}^{2} X_j{}^T \times \mathbf{A} \times X_j = \frac{1}{2} \times (16 + 18) = 17$$

In this example, the total number of edges in G is $\mid E \mid = 20$. Let E_c denote the number of edges cut. We know $E_c + E_{nc} = \mid E \mid = 20$. So, $E_c = 20 - 17 = 3$ that means 3 edges joining nodes in distinct subgraph of the partition. This result is consistent with the cut shown in Figure 8.3.

Example 2: Consider the problem of partitioning the nodes of the graph G of Figure 8.4 into three sets containing 3, 3, 4 nodes respectively. The partition $S_1 \cup S_2 \cup S_3 = \{1,2,3\} \cup \{4,5,10\} \cup \{6,7,8,9\}$ cuts 6 edges and appears to be optimal. For circuit partition, that means nodes 1,2,3 are allocated to subcircuit S_1 and nodes 4,5,10 are allocated to subcircuit S_2 and nodes 6,7,8,9 are allocated to subcircuit S_3. The 6 cutting edges represent the number of interconnections ampng these three subcircuits.

Applying the integer programming analysis to this partition example, the results show that $X_1 = (\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\ )^T$, $X_2 = (\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 1\ )^T$ and $X_3 = (\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 1,\ 0\ )^T$. E_{nc} is calculated in the following:

$$E_{nc} = \frac{1}{2}\sum_{j=1}^{3} X_j{}^T \times \mathbf{A} \times X_j = \frac{1}{2} \times (8 + 6 + 14) = 14$$

We know $E_c + E_{nc} = \mid E \mid = 20$. So, $E_c = 20 - 14 = 6$ that means 6 edges joining nodes in distinct subgraph of the partition. This result is consistent with the cut shown in Figure 8.4.

8.3 Computing Lower Bound on the Number of Interconnections

The following is the analysis of finding an upper bound on the value of the maximum in the partitioning equation discussed in above. Let $V_j = \frac{X_j}{\sqrt{m_j}}$, $j = 1, 2, ..., k$. According to the definition of X_j, the $V_j's$ form an orthonormal set of vectors. Let $\lambda_1 \geq \lambda_2 \geq .... \geq \lambda_n$ denote the eigenvalues of adjacency matrix **A** and let U_1 , ..., U_n denote a corresponding orthonormal set of eigenvectors. Then,

$$A[\ U_1\ ,\ U_2\ ,\ ...,\ U_n\ ] = [\ \lambda_1\ U_1\ ,\ \lambda_2\ U_2\ ,\ ....,\ \lambda_n\ U_n\ ]$$

$$= [\ U_1\ ,\ U_2\ ,\ ...,\ U_n\ ] \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}$$

Assume $P = [\ U_1\ ,\ U_2\ ,\ ...,\ U_n\ ]$. Then, $P^T \times P = I$. Thus, $P^{-1} = P^T$ and therefore,

$$A = [\ U_1\ ,\ U_2\ ,\ ...,\ U_n\ ] \times \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix} \times \left[\ U_1^T U_2^T : U_n^T\ \right]$$

Thus,

$$A = \sum_{i=1}^{n} \lambda_i \times U_i \times U_i^T \tag{8.2}$$

It follows that

$$\sum_{j=1}^{k} {X_j}^T \times \mathbf{A} \times X_j = \sum_{j=1}^{k} m_j\ {V_j}^T\ \mathbf{A}\ V_j = \sum_{i=1}^{n}\sum_{j=1}^{k} \lambda_i\ m_j\ (U_i^T\ V_j)^2 = \sum_{i=1}^{n}\sum_{j=1}^{k} \lambda_i\ m_j\ s_{ij}$$

where $s_{ij} = (U_i^T\ V_j)^2 \geq 0$. So,

$$\sum_{i=1}^{n} s_{ij} = ||\ V_j\ ||^2 = 1, j = 1, ..., k.$$

and

$$\sum_{j=1}^{k} s_{ij} \leq ||\ U_i\ ||^2 = 1, j = 1, ..., n.$$

Thus, an upper bound on the value of the maximum in the partitioning equation can be derived by solving the following linear programming problem.

$$Maximize \sum_{i=1}^{n}\sum_{j=1}^{k} \lambda_i\ m_j\ s_{ij}$$

Subject to

$$\sum_{i=1}^{n} s_{ij} = 1,\ where\ j = 1, 2, ..., k.$$

$$\sum_{j=1}^{k} s_{ij} \leq 1, \; where \; i = 1, 2, ..., n. \tag{8.3}$$

$$s_{ij} \geq 0 \; for \; all \; i \; and \; j.$$

It turns out that the solution of the problem depends on only a few of the eigenvalues (approximately k) of adjacency matrix **A** and it can be obtained by a simple greedy algorithm. Thus, the amount of work required to obtain an upper bound by the above procedure is quite modest. It only involves computing a few eigenvalues of **A**. With the introduction of slack variables the problem of Eq. (8.3) becomes a special case of the well-known transportation problem.

$$Maximize \sum_{i=1}^{n} \sum_{j=1}^{k} c_{ij} \; s_{ij}$$

Sbject to

$$\sum_{j=1}^{k} s_{ij} = a_i, \; where \; i = 1, 2, ..., n.$$

$$\sum_{i=1}^{n} s_{ij} = b_j, \; where \; j = 1, 2, ..., k. \tag{8.4}$$

$$s_{ij} \geq 0 \; for \; all \; i \; and \; j.$$

where $\sum_{i=1}^{n} a_i = \sum_{j=1}^{k} b_j$.

A simple way to obtain an upper bound satisfying all the constraints of Eq. (8.3) is to set $s_{11} = s_{22} = s_{33} = ... = s_{kk} = 1$ and $s_ij = 0$ for $i \neq j$. In this case, the value of maximum in Eq. (8.3) becomes

$$Maximize \sum_{i=1}^{n} \sum_{j=1}^{k} \lambda_i \; m_j \; s_{ij} = \sum_{j=1}^{k} \lambda_j \; m_j \tag{8.5}$$

It follows that

$$E_n c \leq \frac{1}{2} \sum_{j=1}^{k} \lambda_j \; m_j \tag{8.6}$$

We demonstrate the effectiveness of the upper bound of Eq. (8.6) by illustrations of applying it to Examples 1 and 2. For analysis purpose we have computed the first three eigenvalues of the adjacency matrix for the graph in Examples 1 and 2. They are given by

$$\lambda_1 = 4.418 \quad \lambda_2 = 3.076 \quad \lambda_3 = 2.024$$

Consider the problem of partitioning the nodes of the graph G of Figure 8.3 into two sets containing 5 nodes each in Example 1. The partition $S_1 \cup S_2 \cup = \{1, 2, 3, 4, 5\} \cup \{6, 7, 8, 9, 10\}$ cuts 3 edges and appears to be optimal. $Esubnc = 20 - 3 = 17$. Applying Eq. (8.6) to this example, we obtain $E_n c \leq \frac{1}{2} (5 \times 4.418 + 5 \times 3.076) = 18.735$. $E_n c$ is an integer. So, $E_n c \leq 18$ which is very close to the optimal solution, $E_n c = 17$.

Consider the problem of partitioning the nodes of the graph G of Figure 8.4 into three sets containing 3, 3, 4 nodes respectively. The partition $S_1 \cup S_2 \cup S_3 = \{1, 2, 3\} \cup \{4, 5, 10\} \cup$

$\{6, 7, 8, 9\}$ cuts 6 edges and appears to be optimal as shown in Figure 8.4. Therefore, $E_nc = 20 - 6 = 14$. Applying Eq. (6) to this example, we obtain $E_nc \leq \frac{1}{2}$ ($3 \times 4.418 + 3 \times 3.076 + 4 \times 2.024$) $= 15.289$. E_nc is an integer. So, $E_nc \leq 15$ which is very close to the optimal solution, $E_nc = 14$.

Chapter 9

BIST Design Tools for Sequential Circuits: SEQBIST

Random test generation for sequential circuits is more complicated than for combinational circuits because of the following problems:

- Random sequences may fail to properly initialize a circuit that requires a specific initialization sequence.
- Some control inputs, such as clock and reset lines, have much more influence on the behavior of the circuit than other inputs. Allowing these inputs to change randomly as the other ones do, may preclude generating any useful sequences.
- The evaluation of a vector cannot be based only on the number of new faults it detects. A vector that does not detect any new faults, but brings the circuit into a state from which new faults are likely to be detected, should be considered useful.

Most BIST techniques for general sequential logic involve a fundamental trade-off between time and hardware. We can characterize this trade-off most easily by classifying BIST techniques into two categories: *non-scan* and *scan*. In non-scan BIST, we apply a test pattern and capture a response each clock period. In scan BIST, we use scan capability to apply a test pattern and capture a response each scan cycle. A scan cycle is a number of clock cycles required to shift the pattern into a serial scan path or to shift the response out of the serial scan path (whichever is larger). plus one or more normal mode clocks, For example, if a chip containing 1,000 edge-triggered flip-flops has a single, full scan path, scan BIST requires 1,001 clock periods to apply a test pattern and simultaneously observe the response to the previous pattern - roughly 1,000 times slower than the non-scan approach. The two approaches involve distinct hardware structures and trade-offs.

Pseudo-Scan is the methodology used to access the internal flipflops through the MUXes and the test patterns are provided by the SG module. However, Pseudo-Scan test methodology will increase the number of the input to the CUT as the size of the internal flip-flops and latches increase. Furthermore, with a large number of inputs, the test volume generated by an LFSR may not achieve the desirable fault coverage, especially when the number of the internal flip-flops is prohibitively large. On the other hand, the size of the MISR

is another major concern as the number of the internal flipflops increase. The silicon area required by RA&C will be a signature cost.

In this chapter, we introduce several approaches to test sequential circuits without increasing the number of primary inputs and primary outputs. The idea is to access the internal flipflops present in sequential circuit. And the methods to do that include:

1. scan in the test pattern to the internal flipflops. and scan out the responses after one clock cycle (apply the pattern to the CUT).

2. using hardware to alter the content of flipflops so that the next state will not directly depend upon the previous states.

3. use a combination of both methodologies.

These methods have been used to develop a BIST tool for sequential circuits namely SEQBIST. SEQBIST includes the following four design methodologies:

- Circular BIST for sequential circuit, namely CBIST,
- Full scan design for sequential circuit, namely FSCAN,
- Partial scan design for sequential circuit, namely PSCAN,
- Pseudo-scan design with CBIST for sequential circuit, namely PSCBIST.

9.1 Full Scan Design

The controllability and observability problem is severe in sequential circuits as discussed above. Efficient techniques to generate checking sequences for arbitrary circuits are not yet available. The difficulty in finding a checking sequence motivated the search for other methods for setting and observing the states. The scan design is one popular method to achieve this purpose. The basic philosophy behind the scan design is the easy access to some internal points in the circuit. These internal points are used as test points to apply the test vectors and to observe the circuit response.

Although scan design greatly simplifies the testing of sequential circuits, there are some penalties in the use of it.

- *Area overhead.* The modification of the storage element by adding multiplexers at the input of the flip-flops, or adding data and clock inputs to the flip-flops in level sensitive scan design (LSSD), requires extra hardware. To form a shift register during scan mode, the output of each storage element must be routed to the input of the next flip-flop. In some circuits, the routing overhead may be large.
- *Pin Count.* Extra pins are required for the scan inputs. For example, one extra input is needed to control the mode of operation. Here, the scan-in and scan-out can be multiplexed with normal circuit inputs and outputs.

- *Performance* In the scan design, two levels of extra delay are added between the next state output of the combinational circuit and storage elements. The delay is incurred during normal circuit operation as well. Hence, it increases the system clock period. This also decreases the circuit speed during the normal operation. Moreover, due to the information of a scan chain, the fan-out at the outputs of all storage elements is increased by one, thereby slowing the circuit.

- *Test time* Although scan design simplifies the testing process, the test time requirements may be much higher than for a non-scan design. Each test vector application in the scan design requires the entire scan chain be serially loaded with the new state. Hence, the test time is a product of the length of the scan chain and the number of combinational test patterns. If the circuit has a long scan chain, the test time may be very high. This problem is remedied in VTST by the following way. Instead of configuring all the storage elements in the circuit in a single scan chain, these are configured into multiple scan chains. Each scan chain can have its own scan input and output. This reduces the test time because a number of state bits can be loaded in parallel. Multiplexing of input and output pins is done to minimize the pin overhead.

9.2 Partial Scan Design

In the partial scan design, some of the flip-flops are excluded from the scan chains. In full scan, all the storage elements are connected in the scan chain. Therefore, in partial scan there may be some flip-flops in the circuit that cannot be controlled or observed directly in the scan mode. In other words, if all the scan flip-flops are removed and their inputs and outputs are considered the primary inputs and ouputs, the resulting circuit still is sequential. Partial scan provides a continuous range of designs between non-scan and full scan.

The main issue in partial scan is the choice of flip-flops to be included in the scan chain. One possible criterion is circuit performance that precludes the critical path from the scan chain. The other factors in the choice of flip-flops are fault coverage and the length of test patterns. Selected elements are selected to be scanned, such that there are no cycles in the circuit. In general, a large number of storage elements meet the basic criteria to be connected in scan chains. A set with minimum number of flip-flops may be selected to reduce the area overhead. However, flip-flops in the critical path can be left unscanned to limit the performance penalty.

One advantage of partial scan is that the area overhead is less compared to full scan. In some cases, a small reduction in area may make the design acceptable. Partial scan also reduces the performance penalty compared to full scan, which are in the critical path, leaving these flip-flops unscanned, and thus the performance penalty is minimized in partial scan.

9.3 Circular BIST

This section introduces a Built-In Self-Test (BIST) design methodology that can sequentially test large VLSI circuits with very high fault coverage. The proposed techniques, Circular Built-In Self-Test (CBIST) and CBIST with pseudo-partial scan (PPSCAN), are modeled after the principles of the Circular Self-Test Path. The basis of this method is to trade a minimal increase in hardware overhead for a large increase in fault coverage. It will be shown that this technique yields a much higher fault coverage with reasonable time and test vector length when compared to existing sequential test methods.

9.3.1 Introduction

In general, a synchronous sequential circuit can be modeled as an iterative array of combinational logic. At any time, t, the sequential circuit behaves as a combinational circuit with the memory elements holding the previous state of the logic from time $t = t - t_0$. Hence, the detection of faults in a sequential circuit can be viewed as the detection of multiple faults in a combinational network. As the circuit becomes more complex, the task of generating suitable test vectors becomes increasingly more difficult.

The difficulty in generating test vectors for general sequential circuits arises due to the poor controllability and observability of the memory elements, i.e. flip-flops and latches. Although full scan with combinational test generation is viewed as a satisfactory solution to the sequential testing problem, there are several disadvantages of this approach including: the area overhead required by the added hardware, the subsequent lengthening of circuit delay paths caused by the added hardware, and the length of the resulting tests due to extensive serial shifting of patterns and responses. These disadvantages have led to renewed interest in sequential test generation with design for testability in the form of *partial scan* which applies serial scan only to a selected subset of all memory elements. This concept was first introduced in [26] and recently has received increased attention. Two approaches have been implemented. In the first approach [26], scan flip-flops with low testability are selected. A sequential logic test generator is then used to obtain test vectors. The scan overhead and fault coverage depend upon the quality of the testability analysis. The second approach [27] uses functional vectors and a combinational test generator to select flip-flops and generate test vectors only for the faults not detected by the functional vectors. The scan overhead, in this case, may depend on the quality of the functional vectors. Other heuristics [28, 29, 30] for the selection of scan flip-flops are based on the structural analysis of the sequential circuit.

There are other sequential circuit testing techniques that do not model the sequential circuit as an iterative array. One such technique called *FREEZE* [31] utilizes the current state of the memory elements for more than one clock cycle. While the state coverage is decreased, this method has had limited results by specifically targeting faults that are easily detectable in a particular state.

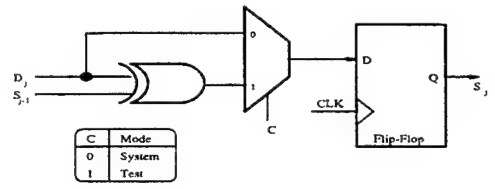


Figure 9.1: CSTP register cell

Most BIST techniques that support the design of self-testing circuits use linear feedback shift registers (LFSR) for test pattern generation and multiple-input signature analyzers (MISR) for test response compaction. With these methods it is necessary that selected conventional registers be replaced with multifunctional registers, such as Built-In Logic Block Observers (BILBO). A technique for designing low cost self-testing VLSI circuits, referred to as *Circular Self-Test Path (CSTP)*, was proposed in [32]. A distinguishing attribute of this technique is low silicon area overhead. The CSTP idea is to chain all input, output, and selected internal registers into a circular self-test path where the next state of the registers depends on the current state of the register as well as the internal combinational logic. In the self-test mode, a selected subset of registers is converted into a cyclic shift register. These registers are modified as shown in Figure 9.1 to act as a MISR in the test mode. The input data is XORed with the contents of the previous shift register bit, which is then shifted into the next shift register bit. The circular self-test path is used to generate test patterns, which are applied to all logic blocks on the chip and the test response is clocked back into the circular self-test path. Because the self-test path works as a MISR, the final content of the MISR is a function of the test vector and the circuit response. In the next cycle, the content of the register is used as a test pattern. This process of simultaneous test pattern generation and compression using the same circular self-test path is repeated for a number of cycles. Finally, the circular self-test path contents are examined to determine whether the circuit has a fault.

From the basic concept and hardware modifications of the CSTP, an efficient built-in self-test methodology for sequential circuit testing has been developed. This method is called *Circular Built-In Self-Test (CBIST)*. Circular BIST provides a low cost BIST alternative to BILBO-based designs since the registers are configured in only two modes (system, test) rather than four (reset, normal, scan, BIST). Another attractive feature of CBIST is that all the functional blocks in the circuit can be tested in one test session. Also, this methodology eliminates the full reliance on testability analysis or functional vectors present in scan-based methodologies.

This section presents the Circular Built-In Self-Test design methodology. Section 9.3.2 discusses the CBIST technique and hardware modifications. Based on the simulation of the ISCAS-89 benchmark circuits, it will be shown that CBIST offers a substantial increase in fault coverage for a moderate increase in hardware overhead. In section 9.4, the CBIST methodology will be enhanced by introducing the concept of Pseudo-Partial Scan. The ISCAS-89 benchmark simulation results of CBIST and CBIST with PPSCAN will be compared to the simulation results obtained from other methods to further show the advantages of the proposed methodology.

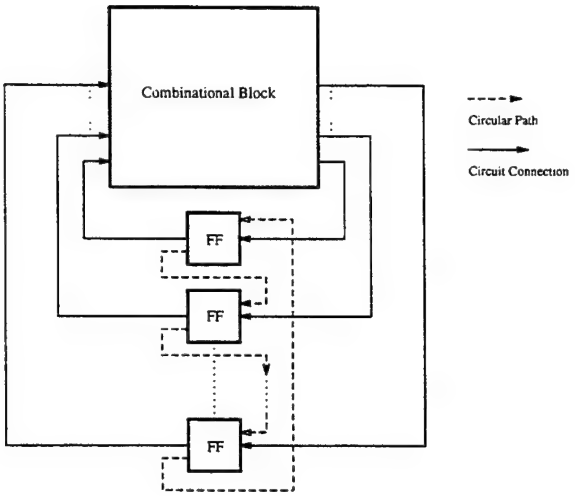


Figure 9.2: CSTP cyclic shift register path

9.3.2 CBIST

A standard technique in Built-In Self-Test methodologies is to employ Linear Feedback Shift Registers (LFSRs) as test pattern generators to generate a pseudo-random set of test vectors. However, while the LFSR generates $2^n - 1$ non-zero and non-repetitive test vectors, the pattern of test vectors will repeat after $2^n - 1$ clock cycles. This is the feature of LFSR test pattern generation that restricts the randomness of the test pattern set and thereby reduces the fault coverage. Circular BIST (CBIST) is designed to increase the randomness of test pattern generation and thus increase the fault coverage.

The basic hardware modification for CBIST implementation concerns the flip-flops or memory storage elements in the sequential circuit design. By modifying the flip-flops as previously shown in Figure 9.1, they may be connected to form a circular chain or path as shown in Figure 9.2. It can easily be seen that the test vector in the circular path is no longer determined solely by the LFSR test pattern generator. It also depends on the functionality of the circuit under test. It was shown in [32] that by making the circular path sufficiently long, an approximately equal probability of a 0 or 1 exists at the register inputs, thus maximizing the randomness of the test pattern generation.

Unlike the CSTP, which uses only the modified flip-flop chain and the circuit under test to generate test vectors, CBIST employs a LFSR test pattern generator and uses the CSTP to increase the randomness of the test pattern generation. While not purely random, the test vectors generated using the CBIST technique yield a much better fault coverage than the test vectors generated from a LFSR test pattern generator alone. In fact, it will be shown that the CBIST technique yields a fault coverage that is very favorably compared with available sequential test methodologies.

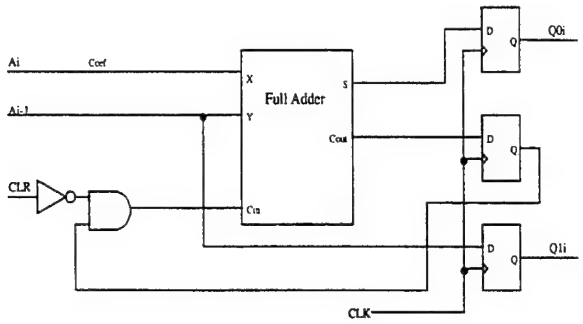


Figure 9.3: Single tree stage

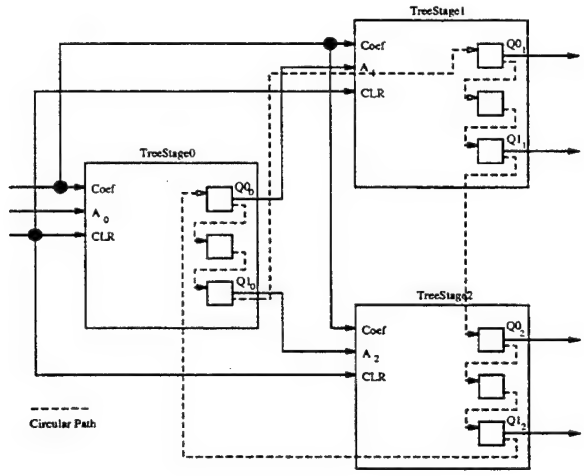


Figure 9.4: Dual tree stage with CSTP

It should be noted that all the registers in the CSTP must be initialized to a known state. This can be accomplished by implementing a clear or set function to the registers. If an exact initial state is needed, scan capabilities can be added at a cost of increased hardware overhead and longer test length.

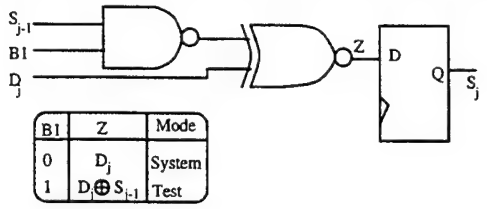
To illustrate the effectiveness of the CBIST methodology, consider the simple sequential circuit in Figure 9.3. Exhaustive testing of the circuit by applying a sequence of seven test vectors generated by a linear feedback shift register yielded a fault coverage of only 79.412% with seven undetected faults. As the tree structure branched to one additional level, as shown in Figure 9.4, the coverage dropped to 70.833% with 28 undetected faults.

Using the CBIST design methodology, the flip-flops of Figures 9.3 and 9.4 were modified as required by Figure 9.1, and a circular path was constructed. Using test patterns generated from a LFSR, the fault coverage rose to 99.315% with a single undetected fault for the DualTree of Figure 9.4 if the flip-flops were reset to zero prior to testing. Furthermore, if the flip-flops were set to one prior to testing, a fault coverage of 100% was attained. The simulation results are summarized in Table 9.1.

The hardware overhead necessary to support the CBIST methodology is essentially one XOR gate and one 2×1 MUX for each register cell in the circular path. However, the cal-

Table 9.1: Fault coverage of linear tree stage

Seq Circuit	Test Method	PI	PO	FF	Faults	Test Pat	FC (%)
Tree	LFSR Only	3	2	3	34	7	79.41
DualTree	LFSR Only	3	4	9	96	7	70.83
Tree	CBIST(CLR)	3	2	3	52	18	100.0
Tree	CBIST(SET)	3	2	3	52	32	98.08
DualTree	CBIST(CLR)	3	4	9	146	32	99.32
DualTree	CBIST(SET)	3	4	9	146	11	100.0



BI	Z	Mode
0	D_j	System
1	$D_j \oplus S_{j-1}$	Test

Figure 9.5: CBIST register cell

culation of hardware overhead by the number of additional gates may be overly optimistic since common implementations of XOR gates and multiplexors require more transistors than other simpler gates.

Another, more accurate, measure of hardware overhead can be obtained using the concept of Cell Units. Using this concept, an inverter is used as the basis for defining the area of larger, more complex gates. An inverter may contain 2 cell units, where an XOR gate may contain 9 cell units. This indicates that the XOR gate consumes approximately $\frac{9}{2} = 4.5$ times as much area as the inverter. For the purpose of calculating the hardware overhead required to implement the CBIST design modifications on the ISCAS-89 benchmark circuits, LSI Logic's LCB007 Cell-Based standard cells were assumed [33]. The cell units for common gates are shown in Table 9.2.

Table 9.2: LSI Logic LCB007 Cell Units for common gates

Logic Gate	Inv	NAND NOR	AND OR	XNOR	XOR	2x1 MUX	DFF w/CLR
C.U.	2	3	4	8	9	9	21

To keep the hardware overhead minimal, the register modifications of Figure 9.1 can be equally represented as shown in Figure 9.5. From Figure 9.1 the hardware overhead can be calculated from as $9 + 9 = 18$ additional cell units where the equivalent representation of Figure 9.5 has an overhead of $3 + 8 = 11$ additional cell units.

Using the cell unit concept, the minimum hardware overhead to implement CBIST in a sequential circuit, containing n flip-flops and C total cell units, can be calculated as

$$\mathrm{HW_{OH}}(\%) = \frac{11n}{C + 11n} \times 100. \tag{9.1}$$

The ISCAS-89 benchmark circuits were modified using the CBIST methodology and were simulated with LFSR generated test patterns with a maximum test set length of 20K patterns. Results show that there is a substantial increase in fault coverage for a moderate

Table 9.3: Circular BIST simulation results for ISCAS-89 benchmarks (20K patterns)

ISCAS bench ckt	CBIST F	CBIST U	CBIST FC (%)	CBIST FE (%)	CBIST HW^1 (%)	CBIST w/ PPSCAN F	CBIST w/ PPSCAN U	CBIST w/ PPSCAN m/N	CBIST w/ PPSCAN FC	CBIST w/ PPSCAN FE (%)	CBIST w/ PPSCAN HW^2 (%)	CBIST w/ PPSCAN HW^3 (%)	FREEZE FC [31] (%)	STG3 FC [34] (%)	PROOFS FC [35] (%)
s298	366	0	100.0	100.0	19.06	362	0	1/14	100.0	100.0	18.86	20.24	65.38	85.7	88.2
s344	418	1	99.76	99.76	17.30	392	0	5/15	100.0	100.0	16.42	22.19	65.00	NA	95.7
s349	426	3	99.30	99.76	17.19	400	2	5/15	99.50	100.0	16.32	22.06	57.34	95.7	95.1
s382	495	0	100.0	100.0	20.59	423	0	9/21	100.0	100.0	19.29	27.56	NA	NA	94.5
s400	520	6	98.85	100.0	20.30	448	6	9/21	98.66	100.0	19.02	27.21	NA	NA	92.7
s444	570	14	97.54	100.0	19.58	498	14	9/21	97.19	100.0	18.33	26.32	NA	89.4	92.7
s510	588	26	95.59	95.59	7.71	568	0	5/6	100.0	100.0	6.62	13.76	NA	NA	NA
s526	641	6	99.06	99.22	18.00	629	3	3/21	99.52	99.68	17.62	20.24	NA	75.3	80.5
s526n	639	2	99.69	99.69	18.00	627	2	3/21	99.68	99.68	17.62	20.24	NA	NA	80.7
s641	581	8	98.62	98.62	13.26	511	3	7/19	98.43	98.43	12.48	17.65	60.79	86.3	87.3
s713	695	45	93.53	98.93	12.71	629	46	7/19	92.69	98.65	11.96	16.96	70.05	80.9	83.0
s820	870	23	97.36	97.36	4.87	854	13	4/5	98.48	98.48	4.19	8.74	53.64	NA	86.5
s832	890	38	95.73	97.26	4.83	874	27	4/5	96.91	98.49	4.16	8.68	53.10	81.4	86.8
s838	985	100	89.85	89.85	16.61	NA	NA	NA	NA	NA	NA	NA	NA	29.6	31.3
s953	1241	30	97.58	97.58	15.09	1083	2	5/29	99.82	99.82	14.69	17.43	8.87	7.8	7.7
s1196	1338	5	99.62	99.62	8.84	NA	NA	NA	NA	NA	NA	NA	99.13	99.7	100.0
s1238	1449	74	94.89	99.64	8.74	NA	NA	NA	NA	NA	NA	NA	91.23	94.7	96.7
s1423	1835	16	99.13	99.89	18.06	1717	19	25/74	98.89	99.71	17.15	23.18	53.67	NA	62.5
s1488	1510	172	88.61	88.61	2.54	1490	12	5/6	99.20	99.20	2.16	4.74	79.37	92.6	94.5
s1494	1530	140	90.85	91.57	2.54	1510	24	5/6	98.41	99.20	2.16	4.73	NA	91.1	93.6
s5378	5397	655	87.86	88.52	15.72	4603	480	33/179	89.57	90.36	15.27	18.30	59.47	74.0	77.2
s9234	8001	2487	68.92	72.67	11.45	7359	1031	54/228	85.99	91.10	11.01	13.99	0.26	0.2	NA
s13207	13167	4237	67.82	68.60	17.73	9731	1192	60/669	89.09	89.12	17.49	19.13	5.49	5.9	NA
s15850	13767	944	93.14	95.84	14.78	13223	1772	95/597	86.60	89.12	14.42	16.91	NA	0.7	NA
s35932	46004	3984	91.34	100.0	18.13	NA	NA	NA	NA	NA	NA	NA	85.32	88.0	88.7
Avg.	4157	521	93.79	95.15	13.79	2287	474	17/152	96.60	97.67	13.20	17.63	56.76	65.50	77.42

F = Total # faults in circuit U = # undetected faults in circuit N = Total # FF's in circuit m = # of FF's selected for PPSCAN

FC = Fault Coverage = $\frac{\text{\# Faults Detected}}{\text{Total \# Faults}}$ FE = Fault Efficiency = $\frac{\text{\# Faults Detected}}{\text{Total \# Faults - \# Redundant Faults}}$ NA = Simulation result not available

HW^1 = Hardware overhead determined by eqn (9.1) HW^2 = Hardware overhead determined by eqn (9.2) HW^3 = Hardware overhead determined by eqn (9.3)

increase in hardware overhead as determined from (9.1). For the benchmark circuits, the average fault coverage was 93.79% and the average hardware overhead 13.79%. The results are tabulated in Table 9.3.

9.4 CBIST with Pseudo-Partial Scan (CBIST w. PP-SCAN)

The fault coverage of sequential circuits using the CBIST methodology can be improved further by the addition of *Pseudo-Partial Scan (PPSCAN).*

Consider a sequential circuit with k primary inputs and l primary outputs containing N flip-flops. The flip-flop selection algorithm presented in [30] is utilized to select a minimum subset of m flip-flops to break feedback cycles of length greater than one. The remaining n flip-flops are modified into CBIST cells and connected in a circular path as described in the previous section. The resulting circuit, after breaking the m cycles, has $k+m$ inputs (k primary inputs and m pseudo-primary inputs), and $l+m$ outputs, as shown in Figure 9.6. By multiplexing the m pseudo-primary inputs with the additional m bits of a $k+m$-bit LFSR test pattern generator, the entire circuit can be tested in one test session through an $l+m$-bit MISR, without serially scanning test patterns into the selected set of m flip-flops. Note that in the real implementation of CBIST in VLSI designs the LFSR and MISR shown in the Figure 9.6 are actually the CBIST register cells of other circuits under test in the same chip and are connected with other CBIST register cells by a circular self-test path. The circular self-test path is used to generate test patterns, which are applied to all logic

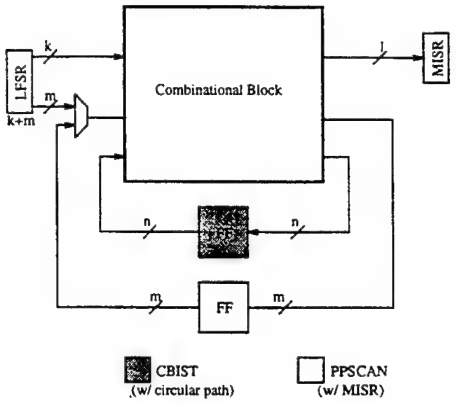


Figure 9.6: CBIST w. PPSCAN

blocks on the chip and the test response is clocked back into the circular self-test path working as a MISR. Therefore the LFSR and MISR are not considered in the following hardware overhead calculation. The hardware overhead necessary to implement CBIST with PPSCAN for a sequential circuit of C cell units can be calculated as

$$\mathrm{HW_{OH}}(\%) = \frac{11n + 9m}{C + (11n + 9m)} \times 100. \tag{9.2}$$

It should be noted that while the internal hardware overhead of (9.2) is less than that of (9.1), "externally" m bits must be added both to the LFSR test pattern generator feeding the $k + m$ inputs and to the MISR receiving the $l + m$ outputs. Since the m selected flip-flops are not being utilized in the circular path or a scan path, they may be modified to act as the additional m-bits of the output MISR, thus saving overall chip-area. Using LSI Logic's LCB007 cell units, a D-type flip-flop can be modified into a MISR cell at a cost of 14 cell units. Equation (9.2) can be modified to represent the hardware overhead including the efficient utilization of the m flip-flops as MISR cells. Equation (9.2) is then given as

$$\mathrm{HW_{OH}}(\%) = \frac{11n + 23m}{C + (11n + 23m)} \times 100. \tag{9.3}$$

The simulation results of the ISCAS-89 benchmark circuits using CBIST with PPSCAN with a maximum of 20K test patterns are tabulated in Table 9.3. Note no flip-flops are selected as pseudo-partial scan flip-flops for s838, s1196 and s1238 circuits. CBIST results show very high fault coverage, especially for the larger benchmark circuits such as s9234 (85.99%), s13207 (89.09%), s15850 (95.84%), and s35932 (91.34%) compared with those produced by three previously proposed sequential testing methods ("FREEZE"[6], "STG3"[9] and "PROOFS"[10]). The high fault coverage is directly related to the efficiency of the CBIST methodology in detecting all nonredundant faults, as evidenced by the *fault efficiency* in Table 9.3. Again, it should be noted that these results were obtained using a *maximum* of 20K LFSR test patterns. It is expected that the fault coverage would improve further as the number of test patterns increases (i.e. state coverage increases).

9.4.1 Conclusions

An efficient BIST approach for sequential circuit testing called *Circular BIST (CBIST)* was presented. Based on the simulation results of the ISCAS-89 benchmark circuits, CBIST and it's derivative *CBIST* with *Pseudo-Partial Scan (PPSCAN)* were shown to have exceptional fault coverage, and a high fault efficiency, at the cost of an average 13.79% (CBIST) and 17.63% (CBISTwPPSCAN) increase in hardware overhead.

Chapter 10

Random Memory Testing

10.1 Introduction

In a widely used fault model for RAM devices, a circuit is divided into three blocks, i.e., memory cell array, decoder circuit, and the read-write circuit. In a memory array, a cell may have a stuck-at 1/0 faults or a cell may have a coupling fault with any other cell. In a decoder circuit, a decoder may not access the addressed cell, it may access a nonaddressed cell or multiple cells. The read-write circuit may have stuck-at 1/0 faults, which appear as memory cell stuck-at faults.

This chapter discusses the problem of testing for stuck-at storage cells in an embedded memory. The following definitions are used in this chapter:

Definition 1: The *error latency* EL_i of a fault f_i is the number of random vectors applied to a circuit until the fault is detected at one of its primary outputs.

Definition 2: The *detection probability per pattern* q_i, of the fault f_i, is the probability that a randomly selected input vector will detect the fault.

Theorem 1: The error latency of a fault has a geometric distribution.

The proof of this theorem has been adapted from [52]. Application of vectors is an independent process. Let the success probability of a Bernoulli trial be q_i. For error latency to assume the value r, the previous r-1 trials should have resulted in a failure and the rth one in success. Thus,

$$Pr\{EL_i = r\} = (1 - q_i)^{r-1} q_i \tag{10.1}$$

The cumulative distribution function of the error latency $F_{EL_i}(t)$ is the probability that the fault is detected at or before input vector t. The *cumulative detection probability* (cdp)

can be obtained from the above equation.

$$F_{EL_i} = Pr\{EL_i \leq t\} = \sum_{r=1}^{t}(1-q_i)^{r-1}q_i$$

$$F_{EL_i} = 1-(1-q_i)^t, t \geq 1 \tag{10.2}$$

Definition 3: The escape probability of a fault f_i is the probability that the fault will go undetected after application of t random input vectors. The escape probability of the cdp adds up to one. The higher the cdp the less likely it is that the faults will not be detected. The smaller the escape probability the better the test quality. The escape probability of a fault f_i is given by

$$Pr\{escape\} = 1 - F_{EL_i}(t) = (1-q_i)^t \tag{10.3}$$

The random test length required to detect a fault f_i with an escape probability no larger than a given threshold e_i can be obtained from the preceding equation as

$$T_i = \left\lceil \frac{\log e_i}{\log(1-q_i)} \right\rceil \tag{10.4}$$

The concepts and definitions developed in this section will be utilized to find the effectiveness of random pattern testing of RAM.

10.2 Random Testing for Stuck-at Faults

This section focuses on finding the stuck-at faults in embedded memories. The hardest fault to detect is the one that affects single bit storage. Figure 10.1 shows the memory and its embedded logic. There are m address lines and n data lines. There exists a single read/write control line. The m input 2^m output decoder is considered to be internal to the memory.

Assumptions

1. Reading from the memory does not destroy its content.
2. There are no sequential circuits preceding the memory.
3. All data lines have equal probability and are independent.
4. The read and write signals are mutually exclusive.

Let the probability of writing into the memory be p_c. Then the probability of reading from the memory is 1 - p_c. Let p_a and p_d be the signal probability of the address lines and the data lines respectively [52].
The probability r of selecting a memory address that has h 1's and m-h 0's is
$r = p_a^h(1-p_a)^{m-h}$
The probability of writing to this address is $rp_c = p_c p_a^h(1-p_a)^{m-h}$
The probability of writing a 1 to any bit at this address is $rp_c p_d = p_1$
The probability of writing a 0 to any bit at this address is $rp_c(1-p_d) = p_0$

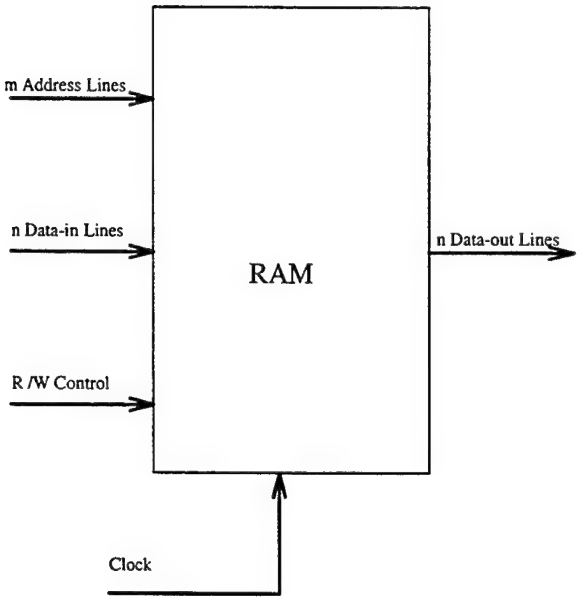


Figure 10.1: An embedded RAM

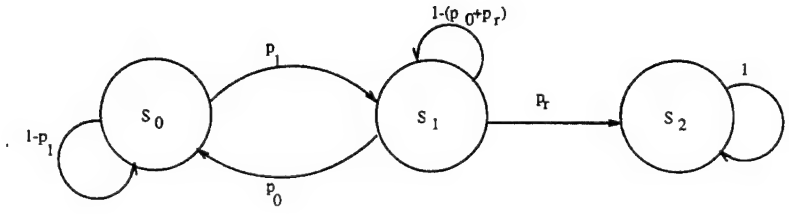


Figure 10.2: Markov model for finding stuck-at-0 faults

The probability of reading from an address is $(1 - p_c)r = p_r$
From the above definition it is evident that $r = p_1 + p_0 + p_r$

Figure 10.2 describes the detection process of a stuck-at-0 faults.

Let the memory cell be initialized to all 1's with the probability i_0. Thus the initial state probabilities are $S_0(0) = 1 - i_0; S_1(0) = i_0; S_2(0) = 0$. The state S_0 represents the storage of logical 0 in the cell. State S_1 represents the storage of logical 1 and is entered by writing a 1 when in state S_0. State S_2 is called the absorbing state and it is entered on a read operation from state S_1.
The difference equations describing the Markov chain are given below

$$S_0(t) = (1 - p_1)S_0(t-1) + p_0 S_1(t-1) \tag{10.5}$$

$$S_1(t) = p_1 S_0(t-1) + (1 - p_0 - pr)S_1(t) \tag{10.6}$$

$$S_2(t) = p_r S_1(t-1) + S_2(t-1) \tag{10.7}$$

A z transform method of solving the difference equation is given below. Modifying the equations we obtain

$$S_0(t+1) = (1 - p_1)S_0(t) + p_0 S_1(t) \tag{10.8}$$

$$S_1(t+1) = p_1 S_0(t) + (1 - p_0 - p_r)S_1(t) \tag{10.9}$$

$$S_2(t+1) = p_r S_1(t) + S_2(t) \tag{10.10}$$

Taking the Z transform we get

$$zS_0(z) - S_0(0) = (1 - p_1)S_0(z) + p_0 S_1(z) \tag{10.11}$$

$$zS_1(z) - S_1(0) = p_1 S_0(z) + (1 - p_0 - p_r)S_1(z) \tag{10.12}$$

We know $S_0(0) = 1 - i_0$ and $S_1(0) = i_0$. From (10.7),

$$S_1(z) = \frac{(z - 1 + p_1)S_0(z)}{p_0} - \frac{(1 - i_0)}{p_0} \tag{10.13}$$

From (10.8),

$$S_1(z) = \frac{p_1 S_0(z)}{(z - (1 - p_0 - p_r))} + \frac{i_0}{(z - (1 - p_0 - p_r))} \tag{10.14}$$

Using the above two equations $S_0(z)$ can be solved for

$$S_0(z) = \frac{z(1 - i_0) + i_0(1 - p_r) - (1 - p_0 - p_r)}{z^2 - (2 - r)z + (1 - p_1)(1 - p_0 - p_r) - p_0 p_1} \tag{10.15}$$

From (10.13) we get

$$S_1(z) = \frac{(z - (1 - p_1))S_0(z)}{p_0} + \frac{(i_0 - 1)}{p_0} \tag{10.16}$$

From (10.10) we know that

$$zS_2(z) - S_2(0) = p_r S_1(z) + S_2(z) \tag{10.17}$$

We also know that $S_2(0) = 0$ from the initialization conditions. Therefore

$$S_2(z) = \frac{p_r S_1(z)}{(z-1)} \tag{10.18}$$

Plugging the value of $S_1(z)$ we obtain

$$S_2(z) = \frac{(z-(1-p_1))(z(1-i_0)+i_0(1-p_r)-(1-p_0-p_r))p_r}{p_0(z^2-(2-r)z+(1-p_1)(1-p_0-p_r)-p_1p_0)(z-1)} + \frac{(i_0-1)p_r}{p_0(z-1)} \tag{10.19}$$

The $S_2(z)$ can be expressed as follows

$$S_2(z) = \frac{\delta_1}{z-1} + \frac{\delta_2}{z-\alpha} + \frac{\delta_3}{z-\beta} \tag{10.20}$$

where α and β are the roots of the following quadratic equation

$$z^2-(2-r)z+(1-p_1)(1-p_0-p_r)-p_1p_0 \tag{10.21}$$

$$\alpha,\beta = \frac{(2-r) \pm \sqrt{(2-r)^2-4[(1-p_1)(1-p_0-p_r)-p_0p_1]}}{2} \tag{10.22}$$

$$= 1-\frac{r}{2} \pm \frac{\sqrt{r^2-4p_1p_r}}{2} \tag{10.23}$$

Define A=$1-\frac{r}{2}$, B =$\frac{\sqrt{r^2-4p_1p_r}}{2}$, $\alpha = A-B$, $\beta = A+B$
Comparing the coefficients of (10.19) and (10.20) we find that
$\delta_2 = \frac{1-A-B-p_ri_0}{2B}$, $\delta_3 = \frac{-1+A-B+p_ri_0}{2B}$, $\delta_1 = 1$
Now inverse Z transform of $S_2(z)$ can be taken to find the cumulative detection probability. The cdp of this test of length is given by

$$f_0(t) = 1 - \frac{C_1(A-B)^t + C_2(A+B)^t}{2B} \tag{10.24}$$

where $C_1 = -1+A+B+p_ri_0$, $C_2 = 1-A+B-p_ri_0$
Substituting the values of A, B, C_1, C_2 we get

$$f_0(t) = 1 - \frac{1-2(1-p_c)i_0+\lambda}{2\lambda}(1-\frac{r}{2}+\lambda\frac{r}{2})^t + \frac{1-2(1-p_c)i_0-\lambda}{2\lambda}(1-\frac{r}{2}-\lambda\frac{r}{2})^t \tag{10.25}$$

where $\lambda = \sqrt{1-4p_cp_d(1-p_c)}$
The test length required to detect stuck-at faults can be derived under the assumption that $\lambda \neq 0$ for large test length. The dominant term is now equated to find the test length.

$$f_0(t) = 1 - \frac{1-2(1-p_c)i_0+\lambda}{2\lambda}(1-\frac{r}{2}+\lambda\frac{r}{2})^t = 1-e_{th} \tag{10.26}$$

Solving for the value of T_0 we obtain

$$T_0 \approx \left\lceil \frac{\ln[\frac{2\lambda e_{th}}{1+\lambda-2(1-p_c)i_0}]}{\ln[\frac{2-(1-\lambda)r}{2}]} \right\rceil \tag{10.27}$$

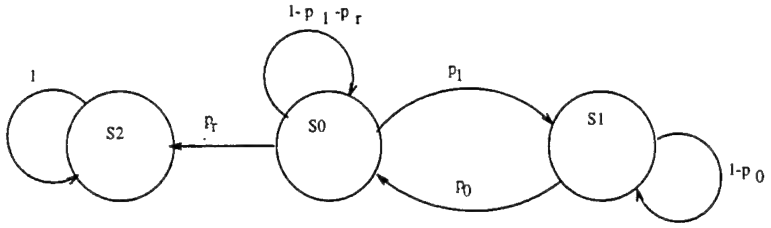


Figure 10.3: Markov model for detecting stuck-at-1 faults

Figure 10.3 is the Markov model for detecting stuck-at-1 faults. Once again the difference equations can be written and solved using z transforms. The various equations describing the states have been given below.

$$S_2(t) = S_2(t-1) + p_r S_0(t-1) \tag{10.28}$$

$$S_0(t) = S_1(t-1)p_0 + [1-(p_1+p_r)]S_0(t-1) \tag{10.29}$$

$$S_1(t) = S_1(t-1)[1-p_0] + p_1 S_0(t-1) \tag{10.30}$$

Let the memory cells be initialized to all 1's with probability i_0. Thus the initial state probabilities are
$S_0(0) = 1 - i_0, \qquad S_1(0) = i_0, \qquad S_2(0) = 0$
By transforming the above equations we get

$$S_2(t+1) = S_2(t) + p_r S_0(t) \tag{10.31}$$

$$S_0(t+1) = p_0 S_1(t) + [1-(p_1+p_r)]S_0(t) \tag{10.32}$$

$$S_1(t+1) = (1-p_0)S_1(t) + p_1 S_0(t) \tag{10.33}$$

The Z transform of the above equations yields

$$zS_0(z) - S_0(0) = p_0 S_1(z) + [1-(p_1+p_r)]S_0(z) \tag{10.34}$$

$$zS_1(z) - S_1(0) = (1-p_0)S_1(z) + p_1 S_0(z) \tag{10.35}$$

From these equations $S_0(z)$ can be solved. Substituting the value of $S_0(z)$ in the following equation we get the $S_2(z)$.

$$zS_2(z) - S_2(0) = S_2(z) + p_r S_0(z) \tag{10.36}$$

$$\Rightarrow S_2(z) = \frac{(p_0 i_0 + (1-i_0)(z-1+p_0))p_r}{[z-1+p_1+p_r][z-1+p_0] - p_1 p_0 (z-1)} \tag{10.37}$$

$S_2(z)$ can again be expressed as a partial fraction. After expressing it as a partial fraction the inverse z transform was taken to get the $f_0(t)$.

$$f_0(t) = 1 - \frac{1-2(1-p_c)(1-i_0)+\eta}{2\eta}(1-\frac{r}{2}+\eta\frac{r}{2})^t$$

$$+\frac{1-2(1-p_c)(1-i_0)-\eta}{2\eta}(1-\frac{r}{2}-\eta\frac{r}{2})^t \tag{10.38}$$

where $\eta = \sqrt{1 - 4p_c(1-p_d)(1-p_c)}$
The test length required to detect stuck-at faults can be derived under the assumption that $\eta \neq 0$ for large test length. The dominant term is now equated to find the test length. Solving for the value of T_1 we obtain

$$T_1 \approx \left\lceil \frac{\ln[\frac{2\eta e_{th}}{1+\eta-2(1-p_c)(1-i_0)}]}{\ln[\frac{2-(1-\eta)r}{2}]} \right\rceil \tag{10.39}$$

10.2.1 Analysis of the results

The results obtained depend upon a number of variables including the p_d, p_c, memory size and the initialization bit. T_0 and T_1 denote the number of test patterns required to detect the stuck-at-0 and stuck-at-1 faults respectively. For a given RAM size the number of test patterns that shall detect both stuck-at-0 and stuck-at-1 faults is the maximum of T_0 and T_1. Figure 10.4 shows the variation of memory size versus the number of test patterns. It can be seen that the memory size varies linearly with the test pattern. Also, for the given initialization the number of test patterns required to detect the stuck-at-0 faults is more than those required to find the stuck-at-1 faults. Figures 10.5 and 10.6 show the variation of the test length for different fault coverage. It can be seen that as the fault coverage requirements increase the test length also increases. It is evident that the number of test patterns required to obtain a fixed fault coverage percentage is higher for the stuck-at-zero case. The test length is proportional to the logarithm of the escape probability, e_{th}.

The address selection probability merits special attention. If p_a is less than 0.5 then the address which is most difficult to access will be 111...1, and if p_a is more than 0.5 then the address which is most difficult to access will be 000...0. Only if the address probability is 0.5 will all the address be equi-probable and the test length will be the shortest. It can be seen that if the p_a is less than 0.5 then the minimum value of r will be p_a^m. However, if the value of p_a is greater than 0.5 then the corresponding minima for r will be$(1-p_a)^m$. The minimum value of r is chosen and used in the formula to obtain the test length. The influence of p_c on the test length depends on the initialization of memory, on p_d, and whether SA0 or SA1 faults are expected.

To investigate the effect of initialization and p_c, the initial probability i_0 is set to zero and one respectively. The data line probability is kept at .5 and the e_{th} was set to 0.01. From the program, the number of test patterns required to detect both SA0 and SA1 faults were found. The test length was then divided by the number of words and the quotient of such a division is called test length coefficient. Figure 10.7 shows the plot of test length coefficient versus the p_c.

It can be seen that when the cells are initialized to all 0s, SA0 faults are more difficult to detect. Similarly it can be seen from Figure 10.8 that when the cells are initialized to all 1's, SA1 faults are more difficult to determine.

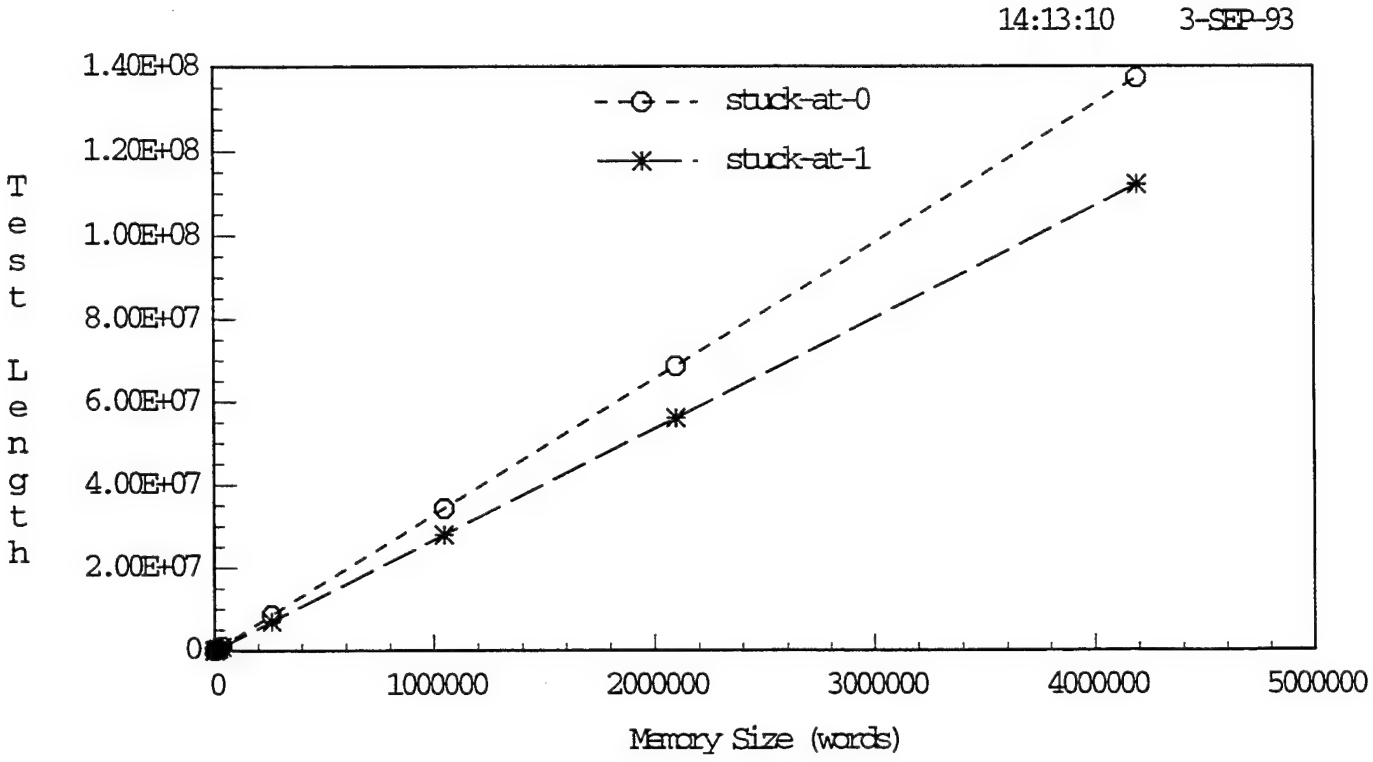


Figure 10.4: Random testing of RAM

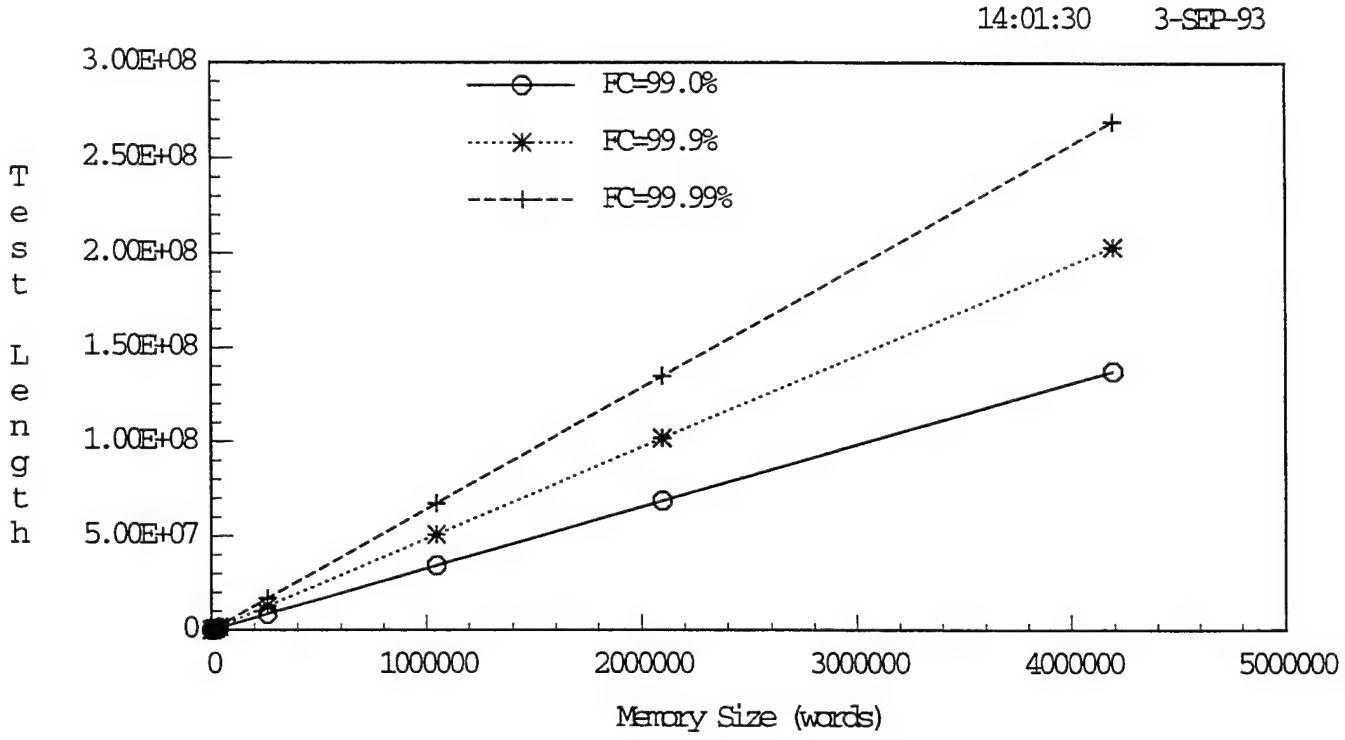


Figure 10.5: Test length for stuck-at-zero vs fault coverage

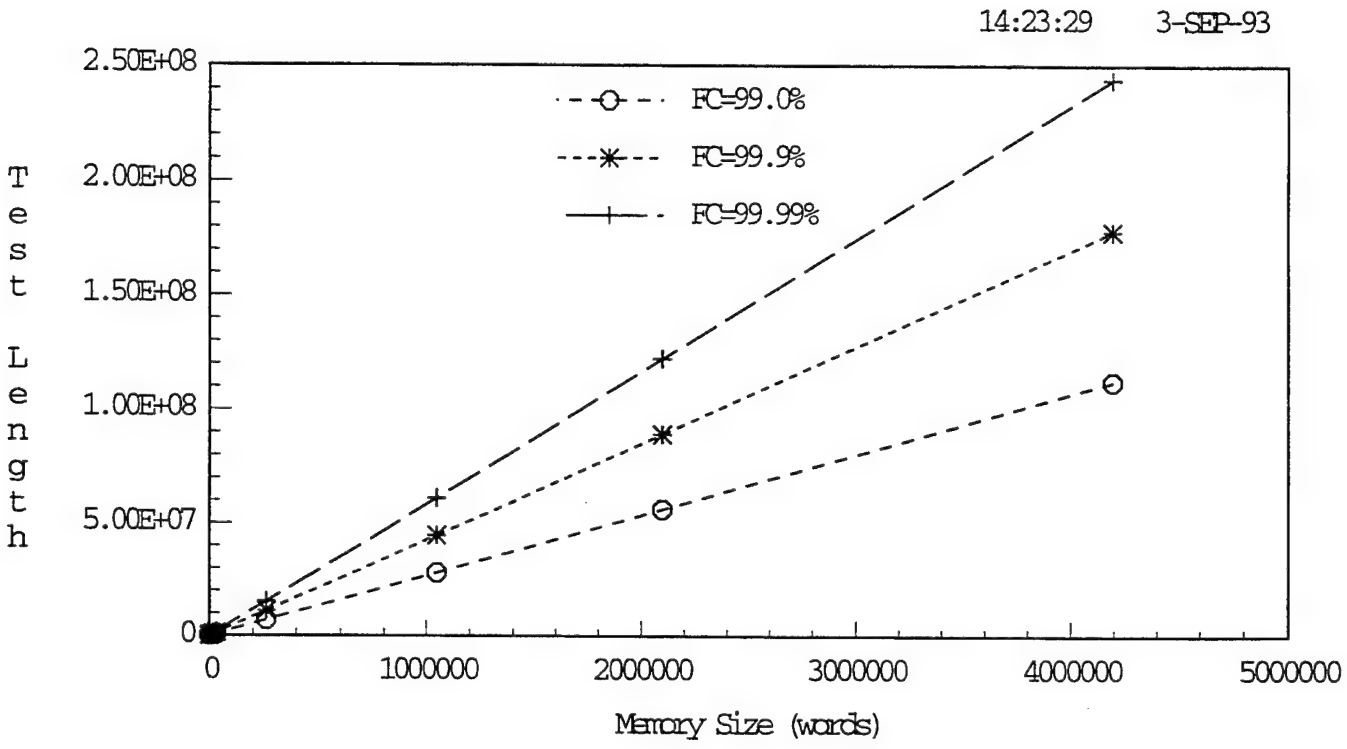


Figure 10.6: Test length for stuck-at-one vs fault coverage

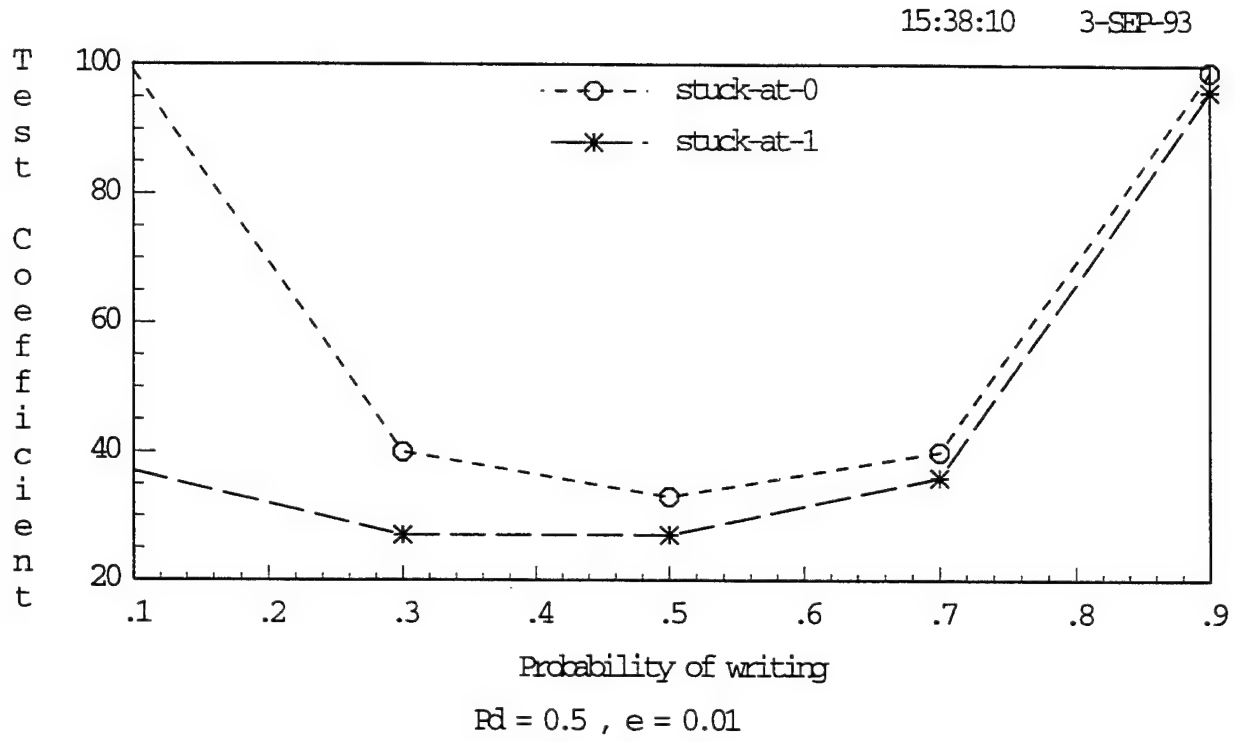


Figure 10.7: Cells initialized to zero

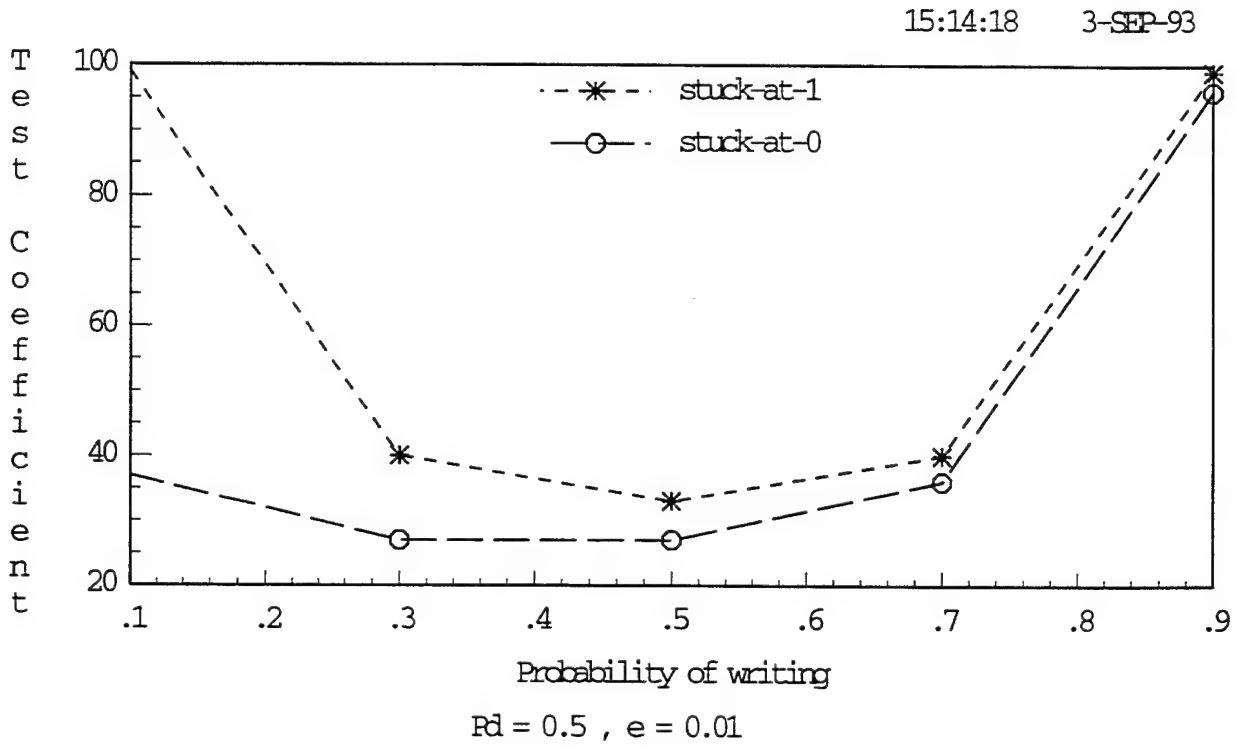


Figure 10.8: Cells initialized to one

The test length is also a function of p_c. In order to detect the SA0 fault a logical 1 has to be written. For low values of p_c the number of operations per cell required is high. Also for large value of p_c the probability of read operation is low. This read operation is necessary to detect the faults. Therefore for higher p_c the test length coefficient is large. It is also clear that the effect of initialization is maximum for short test lengths and decreases rapidly as the test length increases.

10.3 Testing for coupling fault

A coupled fault is a transition creating influence from one storage cell i to another storage cell j, under various conditions relating to the values of the neighboring cells. The model consists of a simple single port n word and each word is 1 bit long. The fault model assumes a coupling such that if there is a 0→1 transition in the influencing cell then the target cell also undergoes the same transition provided some requirements on the neighboring cell states are satisfied. Let G denote some pattern stored in the neighboring cell. Let g_1 and g_2 be the two neighboring cells. A pattern such that g_1 =0 and g_2 =1 shall be represented by $G = \overline{g_1} g_2$. It is apparent that for a n coupling cell fault n-2 neighbors have to be at a specific value.

10.3.1 Random Test Methodology

The memory is initialized correctly to all 1s before the start of the test. Test patterns of the order of 1 million are applied and the responses of the memory to each is compressed in a signature analyzer. Since the write/read signal is driven randomly there is an equal probability of selecting either control signal. Also, the number of trials is very high making the probability of selection of any address, s, equal to $\frac{1}{m}$. The data line signal probability is p. Figure 10.9 has been adapted from [53]. The definitions of the various states are given below

S_{00} : both cells j and i are 0 in good and the faulty memory.

S_{01} : cell j = 0 and cell i = 1 in the good and the faulty memory.

S_{10} : cell j = 1 and cell i = 0 in the good and the faulty memory.

S_{11} : both cell j and i are 1 in the good and faulty memory.

$S_{01/11}$: cells j and i are 0 and 1 in the good memory, and both are 1 in the faulty memory.

$S_{00/10}$: cells j and i are both 0 in the good memory, and are 1 and 0 respectively in the faulty memory.

S_D : the detection state.

Other than the detection state, states $S_{01/11}$ and $S_{00/10}$ are the only possible error states. Each arc of the Figure 10.9 represents the application of a single random vector to the memory. The probability of writing to the memory, w is equal to $\frac{1}{2}$. Let p be the signal probability that a randomly selected input vector will cause a 1 on the line. The difference equations describing the various states are given below

$$S_{00}(t) = (1 - 2swp)S_{00}(t-1) + sw(1-p)S_{01}(t-1) + sw(1-p)S_{10}(t-1) + sw(1-p)S_{00/10}(t-1) \quad (10.40)$$

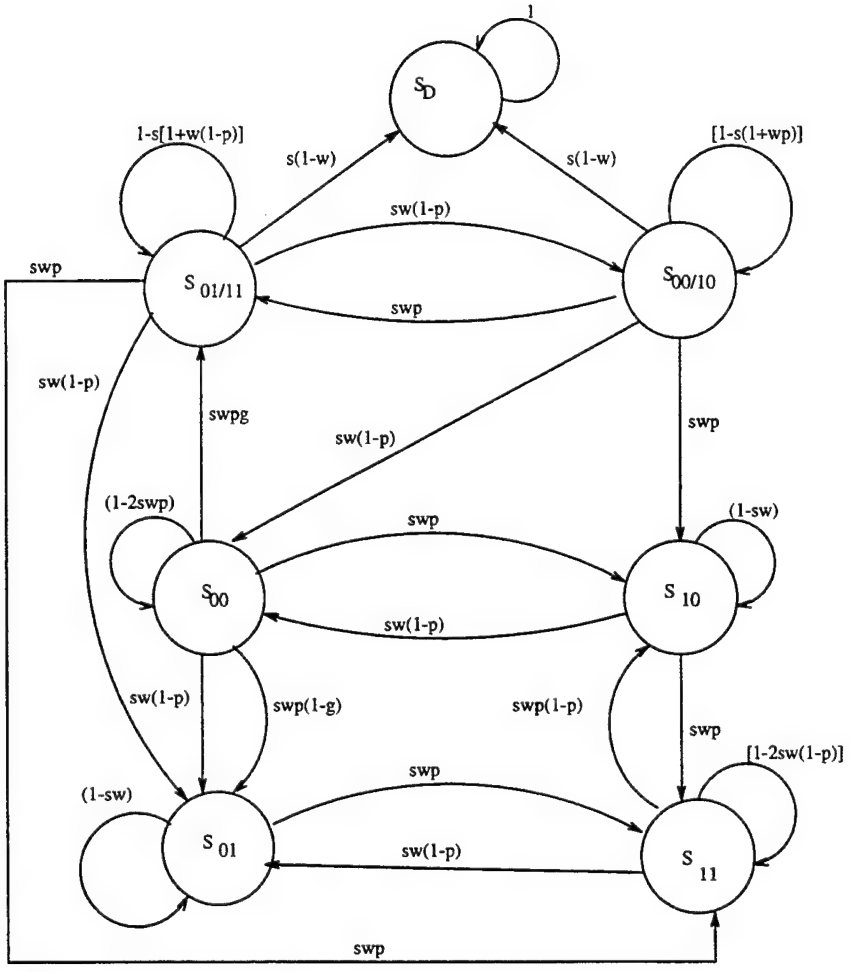


Figure 10.9: Markov chain model for coupling Faults

$$S_{01}(t) = swp(1-g)S_{00}(t-1) + (1-sw)S_{01}(t-1) + sw(1-p)S_{11}(t-1) + sw(1-p)S_{01/11}(t-1) \quad (10.41)$$

$$S_{10}(t) = swpS_{00}(t-1) + (1-sw)S_{10}(t-1) + sw(1-p)S_{11}(t-1) + swpS_{00/10}(t-1) \quad (10.42)$$

$$S_{11}(t) = swpS_{01}(t-1) + swpS_{10}(t-1) + [1-2sw(1-p)]S_{11}(t-1) + swpS_{01/11}(t-1) \quad (10.43)$$

$$S_{01/11}(t) = swpgS_{00}(t-1) + 1 - s[1+w(1-p)]S_{01/11}(t-1) + swpS_{00/10}(t-1) \quad (10.44)$$

$$S_{00/10}(t) = sw(1-p)S_{00/11}(t-1) + [1-s(1+wp)]S_{00/10}(t-1) \quad (10.45)$$

$$S_D(t) = s(1-w)S_{01/11}(t-1) + s(1-w)S_{00/10}(t-1) + S_D(t-1) \quad (10.46)$$

The z transform of each of these equations is taken. Thereafter Mathematica was used to solve for $S_D(z)$. Inverse z transform yields the probability of being in the detection state S_D at time t.

$$S_D(t) = 1 + Ar_1^t + Br_2^t + Cr_3^t + Dr_4^t + Er_5^t + Fr_6^t \quad (10.47)$$

where $r_1 = 1-s, r_2 = 1-\frac{s}{2}, r_3 = 1-\frac{s}{4}[3+\sqrt{\alpha-\beta}]$

$r_4 = 1-\frac{s}{4}[3-\sqrt{\alpha-\beta}]$

$r_5 = 1-\frac{s}{4}[3+\sqrt{\alpha+\beta}]$

$r_6 = 1-\frac{s}{4}[3-\sqrt{\alpha+\beta}]$

$\alpha = 2gp^2 - 2gp + 5$

$\beta = 2\sqrt{g^2p^2(1-p)^2 - 4gp(1-p)(3p-2) + 4}$

This solution is not valid for the case where p is equal to zero or 1.The solution given above is valid only for the range $0 < p < 1$. The boundary conditions used to find the unknown coefficients A B C D E F are : $S_D(0) = S_D(1) = S_D(2) = S_D(3) = 0$ $S_D(4) = \frac{gs^4p(1-p)^2)}{8}$, $S_D(5) = \frac{5gs^4p(1-p)^2(2-s)}{16}$

Let us define the following:

$K = \sqrt{\alpha-\beta}$ $Q = \sqrt{\alpha+\beta}$

The escape probability of the RAM test is given by

$$e_{th} = 1 - S_D(t) \quad (10.48)$$

This can be approximated in the range of interest as $e_{th} \approx -Fr_6^t$ where

$$F = 4\frac{(Q+3)(K^2-9) + 4gp(Q+5)(1-p)^2}{Q(Q^2-1)(Q^2-K^2)} \quad (10.49)$$

Using the equation given above the test length required to detect different coupling faults was determined.

$$t = \frac{\log(\frac{e_{th}}{-F})}{\log r_6} \quad (10.50)$$

In all the calculations the e_{th} was kept at 0.001 and the data-in signal probability was set to 0.5.

Figure 10.10 shows the variation of test length as a function of memory size for different coupling faults. It can be seen from the slope of the line that the test length roughly

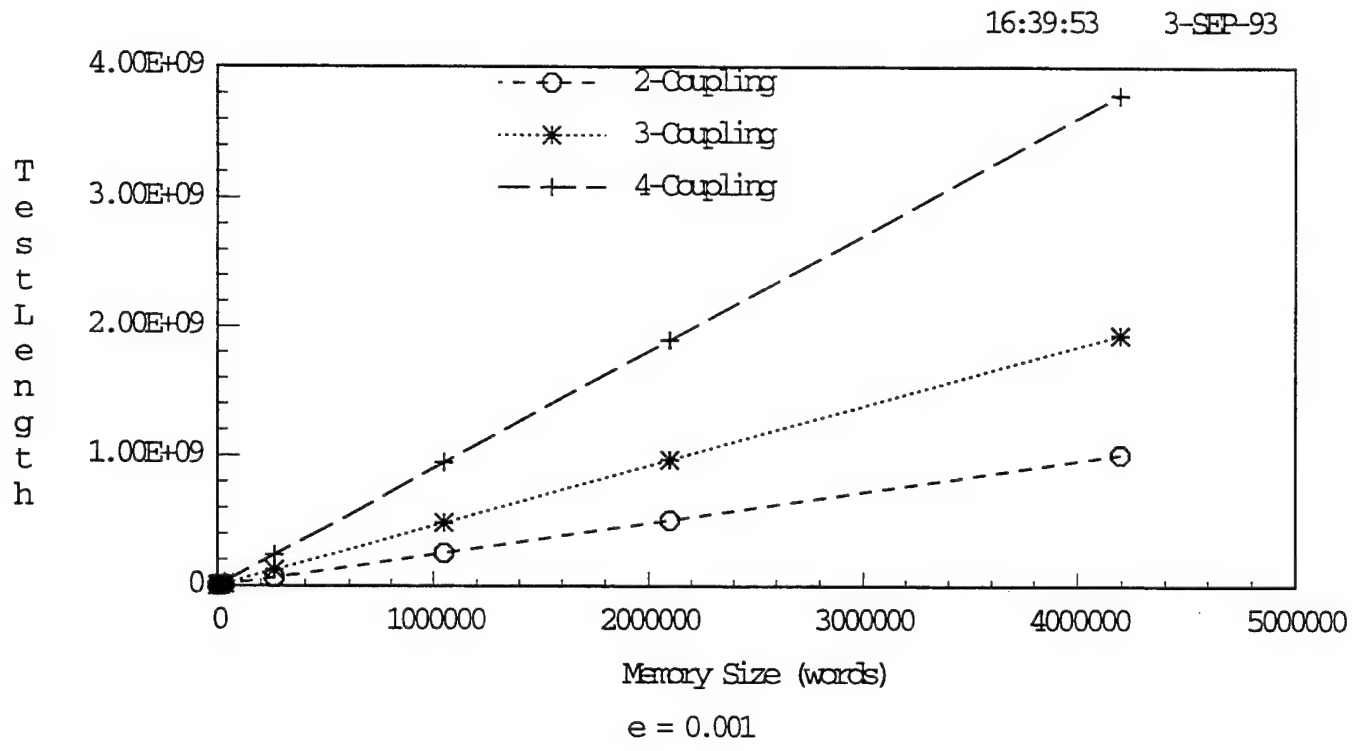


Figure 10.10: Random pattern test length for Coupling Faults

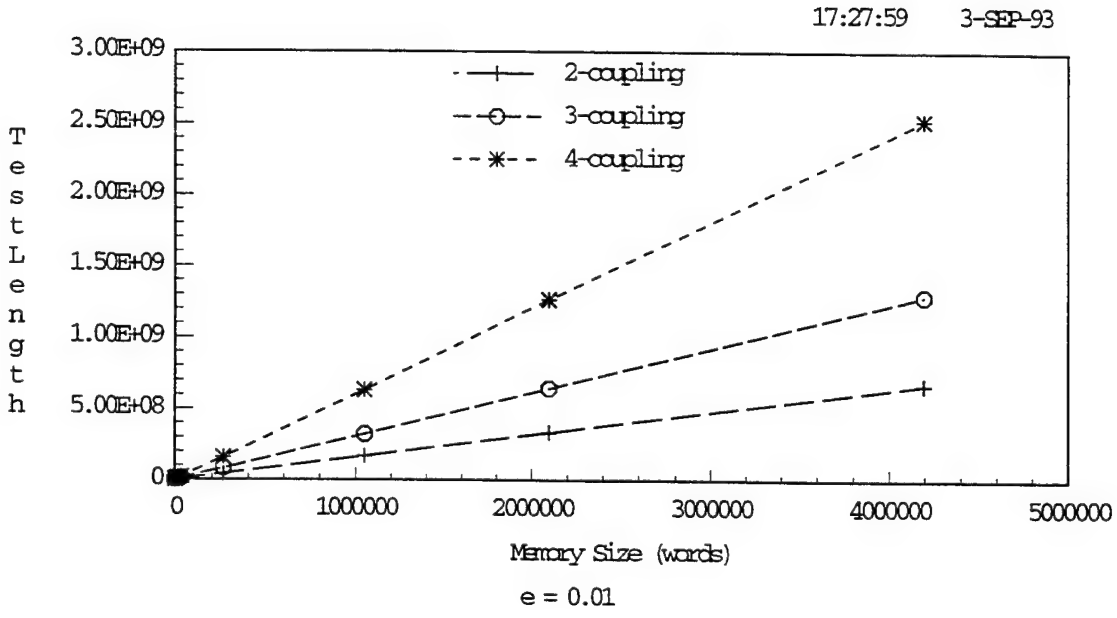


Figure 10.11: Random pattern test length for coupling Faults

doubles when the memory size is doubled for one particular type of fault. The test length also doubles for the same size of the memory when another neighborhood cell is added to it. For a two coupling fault there are no neighborhood cells and a 0 to 1 transition of the influencing cell will cause the target cell to flip from 0 to 1. In the 3-coupling fault it has been assumed that the neighboring cell be at logical one. This implies that the value of g be equal to 1. This occurs with the probability p of the data-in line. In the 4-coupling fault it has been assumed that both the neighboring cells are at logical 1. Mathematically, it means that $g = p^2$. The e_{th} is now changed to 0.01 and for the same values of g and p a graph of test length is plotted in Figure 10.11. It can be seen from the two graphs that the following relationship holds true.

$$\frac{\log T_{e=0.001}}{\log T_{e=0.01}} \approx \log \frac{0.01}{0.001} \tag{10.51}$$

The number of test patterns required to detect the stuck-at 0 faults with 99.99% fault coverage were used to calculate the fault coverage of the coupling faults. The values of g that were used were the same as those used above. It was found that for the 2-coupling faults the fault coverage was constant at 19.9%. For the 3-coupling fault the fault coverage was constant at 10.8%. This was derived under the assumption that g = p. For the 4-coupling fault the fault coverage was found to be 5.65%. In all the cases the fault coverage was independent of the memory size.

10.4 LSI BIST Design

The LSI logic RAM BIST is based upon the principle of bisection of the memory. In the first test cycle all bits of the memory are partitioned into two sections so that no cells of one section are shorted to any cell in another section. In the second test cycle the previously generated halves are further divided into four sections such that no cell in one section is shorted to any cell in another section. The process of bisection stops when no more partitions can be made. In a two cut process, the cuts on the address space are made until the particular address contains only one word. All bits of the single word contain the same binary values. Now the cuts are made on the data space only letting all words be given the same pattern. To detect stuck-at faults complementary test patterns are required. Also each address undergoes a sequential write and read operation. This increases the number of test cycles. A single port RAM with n address lines and m data lines is considered. The number of words therefore is $N = 2^n$. Number of cuts required for the address space is equal to n. Number of cuts required for the data space is q such that $2^q \geq m$. The smallest integer value of q is chosen to satisfy the inequality. The number of total test cycles needed would then be $2 \times (n + q) = t$. Since each address is accessed twice in a test cycle and two dedicated clock signals are needed to write/read an address, the total number of test clock cycles needed in the normal RAM BIST mode would be $2 \times 2 \times N \times t$.

$$T = 2 \times 2 \times 2 \times N \times (n + q) \tag{10.52}$$

$$\Rightarrow T = 8N(\log_2 N + q) \tag{10.53}$$

Assuming a 32 bit word q = 5, we obtain

$$T = 8N \log_2 N + 40N \tag{10.54}$$

10.5 Modified Marching 1/0 Test

Dekker *et al.* [54] have presented a 9N and 13N algorithm for a bit oriented memory. The fault model development divides the SRAM into memory array, address decoder, R/W logic. The defects of the address decoder and the R/W logic are *mapped onto* the functionally equivalents faults in the memory array. The general fault model of the decoder assumes that either an address accesses no cells or more than one cell. These can be viewed as stuck open or multiple access faults. The faults in the read/write logic can be classified as stuck-at, stuck-open and state coupled. A *march element* is a finite sequence of read and/or write operations applied to every cell in the memory array, either in increasing or decreasing address order. A cell exhibits a data retention fault if it fails to retain its logical value after some units of time. The 9N algorithm detects all faults in a bit oriented SRAM with combinational R/W logic. If the R/W logic contains a data latch, detection of stuck-open faults is not guaranteed by the 9N algorithm. A 13N algorithm can be used to detect such faults.

Address	Initialize	March1	March2	March3	March4
0	Wr(0)	Rd(0),Wr(1)	Rd(1),Wr(0)	Rd(0),Wr(1)	Rd(1),Wr(0)
1	Wr(0)	Rd(0),Wr(1)	Rd(1),Wr(0)	—	—
2	Wr(0)	Rd(0),Wr(1)	Rd(1),Wr(0)	—	—
—	—	—	—	Rd(0),Wr(1)	Rd(1),Wr(0)
—	—	—	—	Rd(0),Wr(1)	Rd(1),Wr(0)
N-1	Wr(0)	Rd(0),Wr(1)	Rd(1),Wr(0)	Rd(0),Wr(1)	Rd(1),Wr(0)

Table 10.1: 9N Linear Test Algorithm

The algorithms presented earlier are valid only for a bit oriented memory. For a word oriented memory the read or write operation will include all the bits of a word. An entire word of data is called *data background*.The algorithms described earlier need to be altered to accomodate the word oriented memories. The following instructions need to be redefined:
Wr(0) := Wr(≪data background≫)
Rd(0) := Rd(≪databackground≫)
Wr(1) := Wr(≪inverted data background≫)
Rd(1) := Rd(≪inverted data background≫)

Word oriented SRAMs have the problem of state coupling. To detect that fault all states of two arbitrary cells at the same address must be checked. If m bits per word are used and if m is a multiple of two, then the minimum number of required data backgrounds, K, can be found thus

$$K = \log_2 m + 1 \tag{10.55}$$

Address	Initialize	March1	March2	March3	March4
0	Wr(0)	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)
1	Wr(0)	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)	—	—
2	Wr(0)	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)	—	—
—	—	—	—	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)
—	—	—	—	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)
N-1	Wr(0)	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd(0)	Rd(0),Wr(1),Rd(1)	Rd(1),Wr(0),Rd (0)

Table 10.2: 13N Linear Test Algorithm

Assuming a 32 bit word, we obtain K equal to 6. First the test is run with data background 1, then with data background 2 and so on. The test length is therefore equal to six times 9N or 13N, where N is the number of words. If the data retention test is also included the test length becomes T_1 and T_2 which are defined below.

$$T_1 = 54N + 1 \tag{10.56}$$

$$T_2 = 78N + 1 \tag{10.57}$$

10.6 Transparent BIST RAM

Kebichi *et al.* [55] have proposed a transparent RAM built-in self-test circuit based on the 17N algorithm of Marinescu. In a digital circuit BIST is used periodically to test the RAM. The material in this section is based on [55]. It is useful to save the contents of RAM each time self testing is done. This is a very stringent constraint on the design. The problem is further complicated if the RAM is embedded in some larger digital circuit. Accessing the RAM and saving its content is a difficult task in such a situation. The modified Marinescu algorithm is shown in Table 10.3. In the sequence S1 given above Ra(1) means that a read operation is performed on the cell and the value read from the cell is the initial data in the cell. In all other sequences Ra(1) implies that a read operation is performed and the result is the expected value of the data. Similarly Ra(0) refers to the complement of the data. Wa(1) and Wa(0) indicate writing the data and its complement respectively. A register is required to save the contents of the last read address cell. This value or its inverse is used to perform the next operation on the cell at the same address. An algorithm to predict the signature of the RAM is also implemented as shown in the Table 10.4.

Once the data of these sequences is read, it is injected into the output response compactor. The data read during these sequences is inverted and compared to the data in Table 10.3. The predicted signature is then shifted out or is stored in some register. The

Num	S1	S2	S3	S4
1	Ra(1)Wa(0)Wa(1)Wa(0)	Ra(0)Wa(1)Ra(1)Wa(0)	Ra(0)Wa(1)Wa(0)Wa(1)	Ra(1)Wa(0)Ra(0)Wa(1)
2	Ra(1)Wa(0)Wa(1)Wa(0)	Ra(0)Wa(1)Ra(1)Wa(0)	Ra(0)Wa(1)Wa(0)Wa(1)	Ra(1)Wa(0)Ra(0)Wa(1)
3	Ra(1)Wa(0)Wa(1)Wa(0)	Ra(0)Wa(1)Ra(1)Wa(0)	Ra(0)Wa(1)Wa(0)Wa(1)	Ra(1)Wa(0)Ra(0)Wa(1)
4	Ra(1)Wa(0)Wa(1)Wa(0)	Ra(0)Wa(1)Ra(1)Wa(0)	Ra(0)Wa(1)Wa(0)Wa(1)	Ra(1)Wa(0)Ra(0)Wa(1)
-	Ra(1)Wa(0)Wa(1)Wa(0)	Ra(0)Wa(1)Ra(1)Wa(0)	Ra(0)Wa(1)Wa(0)Wa(1)	Ra(1)Wa(0)Ra(0)Wa(1)
n	Ra(1)Wa(0)Wa(1)Wa(0)	Ra(0)Wa(1)Ra(1)Wa(0)	Ra(0)Wa(1)Wa(0)Wa(1)	Ra(1)Wa(0)Ra(0)Wa(1)

Table 10.3: Modified Marinescu's Algorithm for Transparent BIST

Num	Sa	Sb	Sc	Sd
1	Ra(1)	Ra(1)Ra(1)	Ra(1)	Ra(1)Ra(1)
2	Ra(1)	Ra(1)Ra(1)	Ra(1)	Ra(1)Ra(1)
3	Ra(1)	Ra(1)Ra(1)	Ra(1)	Ra(1)Ra(1)
4	Ra(1)	Ra(1)Ra(1)	Ra(1)	Ra(1)Ra(1)
-	Ra(1)	Ra(1)Ra(1)	Ra(1)	Ra(1)Ra(1)
n	Ra(1)	Ra(1)Ra(1)	Ra(1)	Ra(1)Ra(1)

Table 10.4: Signature Prediction Algorithm

new signature is then compared to the predicted one. Based on Tables 10.3 and 10.4. the test length T can be determined.

$$T = 16N + 6N \tag{10.58}$$

The equation however is true only for a bit oriented SRAM. For a word oriented SRAM the above formula changes.

$$T = [\log_2 m + 1] \times [16N + 6N] \tag{10.59}$$

Assuming a 32 bit word, we obtain

$$T = 132N \tag{10.60}$$

10.6.1 Comparison of BIST Techniques

We will now compare the results of the preceding BIST techniques. Figure 10.12 shows the plot of test length versus the memory size for Transparent BIST, Modified 1/0 and LSI BIST. It is evident that the LSI BIST requires the maximum number of test patterns to detect the faults with a certain degree of confidence. The 13N algorithm requires the least number of test patterns to detect the faults with the same escape probability. The Transparent BIST requires an intermediate test length to achieve the same fault coverage. The plot for LSI BIST exhibits a logarithmic increase that is much higher than any other method. The following equation was solved to find the word size for which the fault coverage given by LSI BIST and Transparent BIST is the same.

$$8N \log_2 N + 40N = 132N \tag{10.61}$$

For $N \neq 0$, we obtain N equal to 2896.30. For any memory size higher than 2896 words the LSI BIST technique will require more patterns to guarantee the same fault coverage. Similarly the test length for LSI BIST and Modified 1/0 techniques is equated to obtain the common word size for the same fault coverage.

$$8N \log_2 N + 40N = 78N + 1 \tag{10.62}$$

This equation was solved using the Newton Raphson numerical method. The value of N converges in the neighborhood of three after one hundred iterations. For a memory size higher than three words the LSI BIST increases very sharply. In fact the test length for LSI BIST is a positively increasing monotonic function. Since the size of memory is usually very large we can neglect 1 in comparison to 78N. Using this approximation we get N = 27.

There exists no word size for which the 13N and Transparent BIST algorithm have the same fault coverage. The transparent BIST has the advantage that the contents of RAM are not destroyed during periodic testing. Figure 10.13 shows the test length comparison of LSI-BIST versus the test length required for coupling faults. More test patterns are required to detect the 3-coupling fault than that required by LSI-BIST for the same fault coverage. Figure 10.14 shows the plot for the test length of the random method and BIST techniques versus the memory size for a 32 bit word. The test length required to achieve 99.99% fault coverage for single stuck-at faults was used. The LSI BIST requires maximum test length and random pattern testing for single stuck-at faults requires the least test length. The test length for two coupling faults has 99.00% fault coverage. The modified 13N and the transparent BIST method require intermediate test length. This figure confirms that pseudo random testing of RAM yield the best test length requirements.

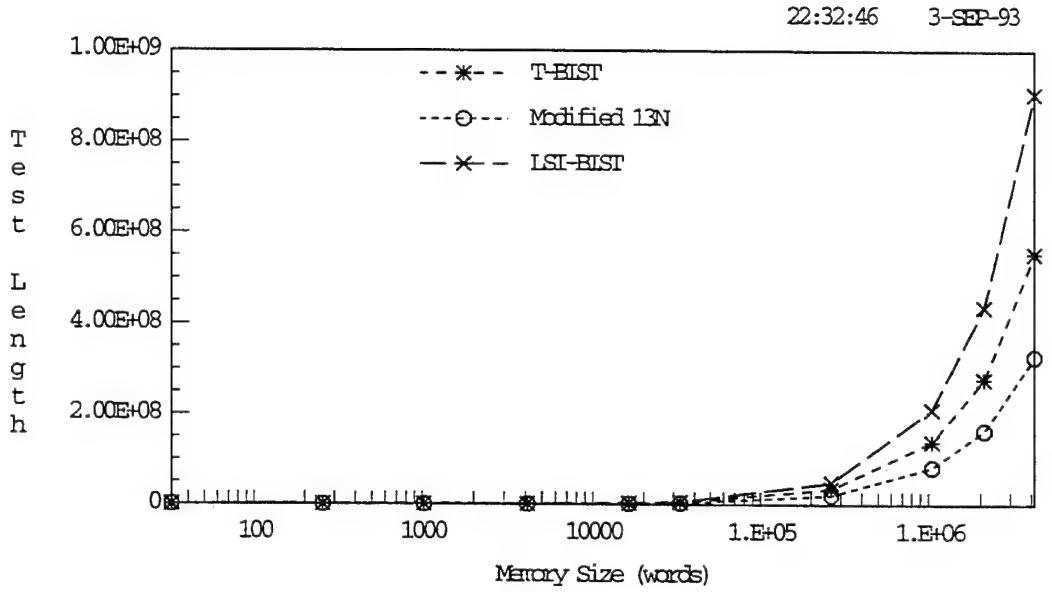


Figure 10.12: Comparison of LSI-BIST test length with T-BIST and 13N

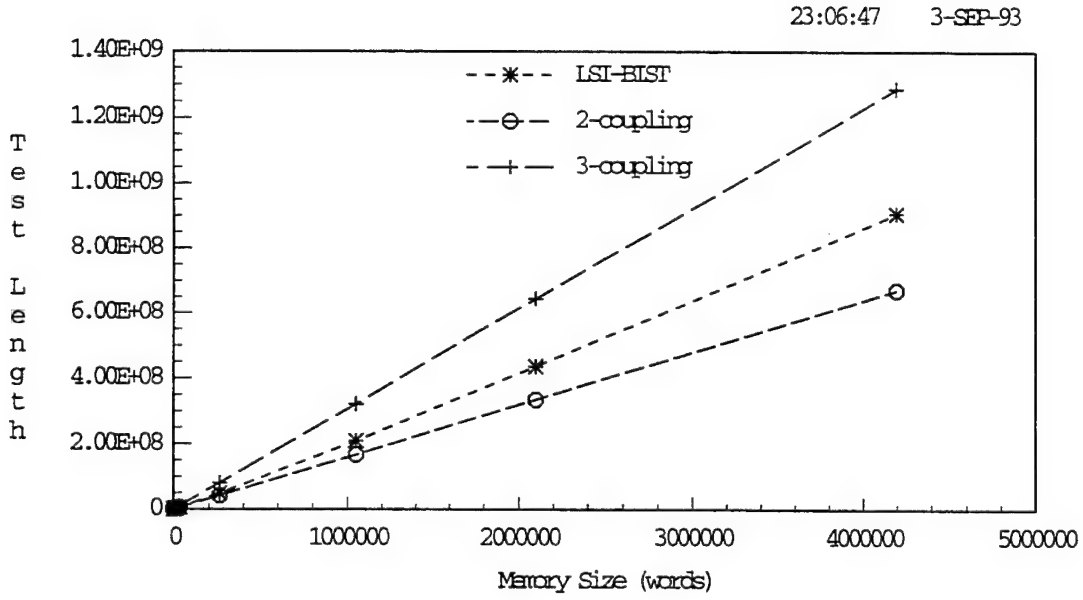


Figure 10.13: Coupling fault versus LSI BIST

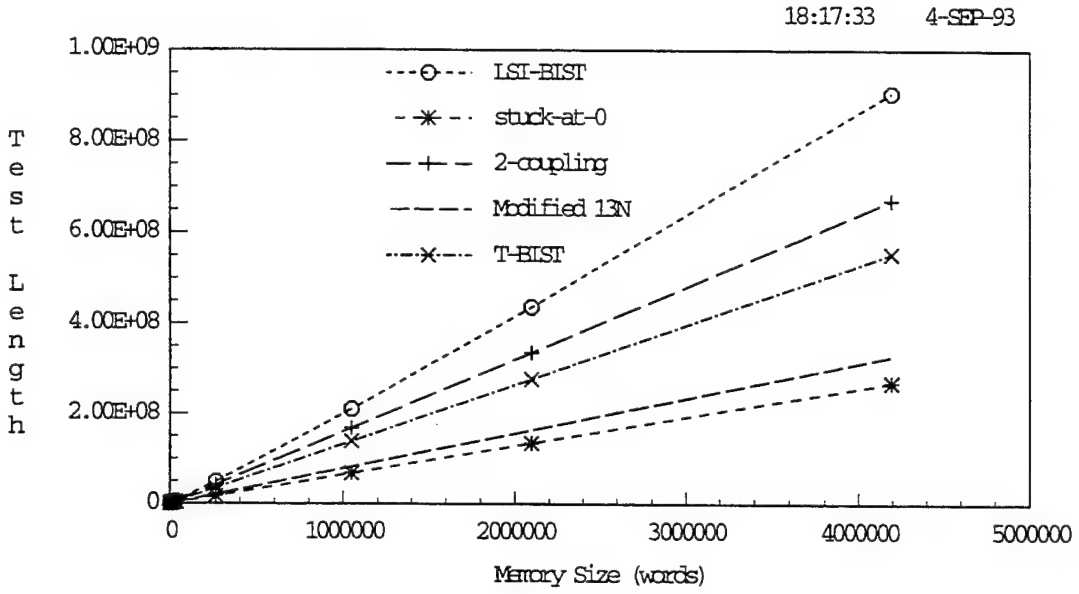


Figure 10.14: Comparison of RAM testing techniques

Chapter 11

NDL BIST Automation Tool: BUILDER

11.1 Introduction

VTST provides all the information that is needed to build a BISTed circuit. The information includes the insertion of test points, BIST components, interconnections among test points, BIST components, primary inputs and primary outputs. Test point insertion, BIST components generation, and interconnections are very time consuming tasks if they are done manually. Moreover, manual design errors may occur during the design phase. Therefore, a BIST automation tool becomes in high demand.

The interface among the NDL Net BUILDER and the VTST synthesis tools is shown in Fig 11.1. The VTST parser tool converts a set of hierarchical NDL netlists to a flat netlist named testability description netlist (TDN). VTST synthesis tools work on TDN netlist and generate the synthesis results called *actions* which include all the necessary information for BUILDER to do BIST insertion.

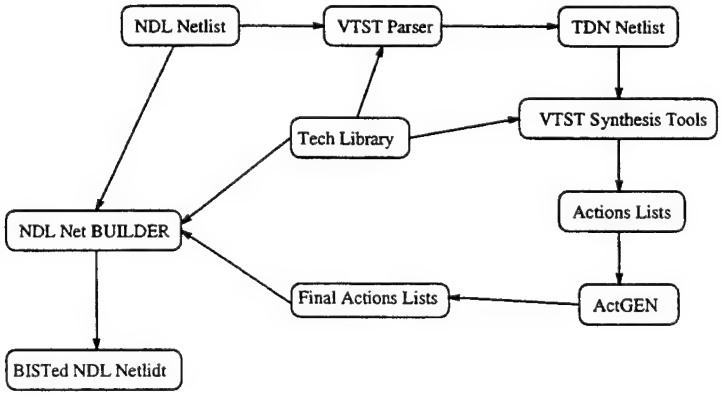


Figure 11.1: VTST and NDL netlist BUILDER

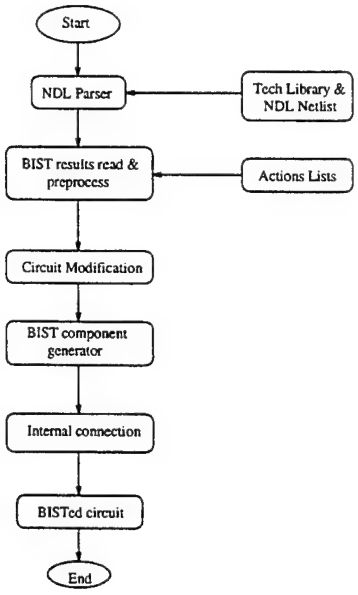


Figure 11.2: NDL Netlist BUILDER

There are two strategies to apply VTST to a circuit. VTST can be applied to the top module or to any selected sub-module under the top module of a CUT. The former strategy generates one final action list for the entire module, the latter one will generate more than one action list because every selected sub-module generates an action list. No matter which strategy is selected, BUILDER needs to read in all of the NDL netlists from the top to the bottom modules to perform the BIST insertion.

11.2 BUILDER

The block diagram of BUILDER is shown in Fig 11.2. Basically, BUILDER first reads the hierarchical NDL netlist, then reads the flat actions and converts those actions back to a hierarchical structure. After that, BUILDER can map the actions to the hierarchical NDL netlist and insert the test points. BUILDER also uses the information from actions lists to generate BIST components such as TPG, MISC, MUX array, XOR tree, etc. After all the test points are inserted and BIST components are generated, BUILDER will make the final connection and print out the BISTed design.

11.2.1 NDL Parser

BUILDER accepts the NDL netlist as its input circuit format. The NDL parser can parse multiple netlist files and each file may contain more than one NDL module. The NDL parser builds a hierarchical data structure to store the NDL netlist in memory. The module's data structures (components) start with the top most module containing input

and output ports, all the instances that compose this module, and internal signal mappings. The instance data structure contains two pointers; one points to the component that this instance belongs to, and the other points to the component that expresses the nature of this instance. It also contains a list of port maps, that is, maps of the actual signal names in this component, and maps of port names on this instance.

11.2.2 BIST Results (Actions) Read and Preprocess

More than one action list will be generated if subcircuits of the CUT select different BIST methodologies during BIST synthesis. Then, at the final stage, the user needs to collect those action lists in which the BIST methodologies were selected. All these action lists will be processed by an actions consolidated utility named *ActGEN* to generate a consolidated action list.

BUILDER will preprocess the actions during the actions reading process. The process includes:

- **Build hierarchical actions list.** As we mentioned before, BISTDaTA works on the TDN format, that is, the actions generated by the synthesis tools are in a flat format. BUILDER's parser creates a hierarchical netlist, so the flat action information needs to be converted back to a hierarchical structure to match BUILDER's internal data structure.

- **Reorganize I/O order.** I/O's order is very important in BIST automation, especially when BISTSYN generator or Autonomous tools are selected. Based on the different I/O related actions, BUILDER will create special a I/O list or map list to store this information.

- **Collect shared signals.** There are two kinds of shared signals. One kind is shared test inputs, and the other is shared control signals. If a shared signal related action is read, BUILDER will check the signal name first. If it is the first time that the name is used, BUILDER will put the name into a queue that can reflect the nature of this shared signal, and process the original action. Otherwise, BUILDER will modify the action to let test point insertion handle the action.

11.2.3 Circuit Modification

BUILDER modifies the hierarchical circuit netlist with BIST insertion according to the sequence of actions. Basically, the modification includes :

- **Test point insertion.** Test point means a node in a circuit at which a test pattern can be injected or the response can be observed. If a desired test point is neither a primary input (PI) nor primary output (PO), BUILDER needs to insert a pseudo

primary input (PPI) or pull out a pseudo primary output (PPO). In most cases, PPI insertion means to add one multiplexer at one test point. The added PPI and PPO will be propagated to an upper level where the corresponding component and instance also need to be updated.

- **Control signal insertion.** Control signal refers to the signals such as clock, clear, set, and enable. During the test mode, all component's control signals must be synchronized and controlled by an external control signal. For this purpose, a multiplexer is inserted in front of the control input of an instance. So during the testing, the control signal directly comes from an external control signal instead of from internal logic.

- **Scan component replacement.** If scan design is involved, BUILDER needs to replace all the D flip-flops (Dff) on the scan path with the equivalent scan Dff. The connection information between scan Dffs in the scan chain will be propagated to the top most level.

- **CBIST component insertion.** If CBIST design is involved, BUILDER will add a mux and an xor gate before the data input port of the desired Dff. Again, the connection information between this xor's input and the previous CBISTed Dff's output will be propagated to the top most level.

- **Signal connection and pin insertion.** After PPI or PPO are inserted, additional ports and port maps are added to the associated component and instance. Again, the added PPI and PPO will be propagated to the upper level where signal connection to other logic are required. These include scan, CBIST, and shared signal connections. Because the connection may be across different levels, all the connections will be initialized at the top most level.

11.2.4 BIST Component Creation

BUILDER can generate the following BIST components :

- **TPG and MISR.** Depending on the number of modules under test, BUILDER can build a fixed or variable size TPG and MISR. The size of a TPG is determined by the maximum dependency value among modules, i.e., BISTSYN generator. The size of a MISR is determined by the maximum output size among all modules but is limited to 64 bits. If any module has output size greater than 64, an XOR tree is inserted to compact the outputs to 64-bit during testing.

- **XOR tree.** XOR tree is used to reduce the module's output size during testing in order to match the MISR's size. BUILDER will check all the modules and generate XOR trees for those modules whose output size are greater than 64 bits.

- **MUX array.** If there are more than one module under test, a MUX array is needed. External module select signals will select which module is currently under test, and therefore control the MUX array to select the module's outputs feeding into the MISR

for signature analysis. There is no MUX array needed, if there is only one module under test.

11.2.5 Internal Connection

After circuit modification and BIST component creation, BUILDER will make the necessary connections among the modified circuit and the BIST components. Basically, there are two major types of connections, one is to send test patterns from the TPG to the module under test; the other is to send the responses from the module under test to the MISR.

- **Send test patterns.** This part refers to the connections between the TPG and the module's input ports. The connections become even more complex when BISTSYN is involved. It means to connect the TPG and BISTSYN generator's input ports first, and then connect BISTSYN generator's output ports to the module's input ports. One thing is very important for all the connections, that is the order of the ports. Because a different order means a different input pattern. During the connection, BUILDER adds multiplexers on the module's input ports to select normal or test modes. For multiple modules under test, BUILDER examines each module's input ports. If the port is shared by other modules, a tri-state buffer will be inserted between the TPG and the module's input ports to avoid conflict.

- **Analyze responses.** This part refers to the connections between the module's output ports and the MISR. Before making any connections, BUILDER will check the module's output size. If it is greater than 64 bits, BUILDER will connect the module's output ports to the input port of the XOR tree's input ports. If there are more than one module under test, BUILDER will connect the module's output ports (or XOR tree's output ports) to a MUX array, and then connect the MUX array's outputs to the MISR. If there is only one module, BUILDER will connect the module's output ports (or XOR tree's output ports) to the MISR directly.

The rest of the connections are related to VTST control signals, such as clock, clear, set, TPG and MISR's function select signals, and module select signals.

11.2.6 BISTed Circuit Generation

The final step of BUILDER is to generate the BISTed circuit. All the modified circuits, created BIST components, and the connections among them are stored in the BUILDER's internal data structure mentioned in the parser section. BUILDER will read in all the information from the data structure and generate the BISTed design in NDL netlist format.

11.3 Usage

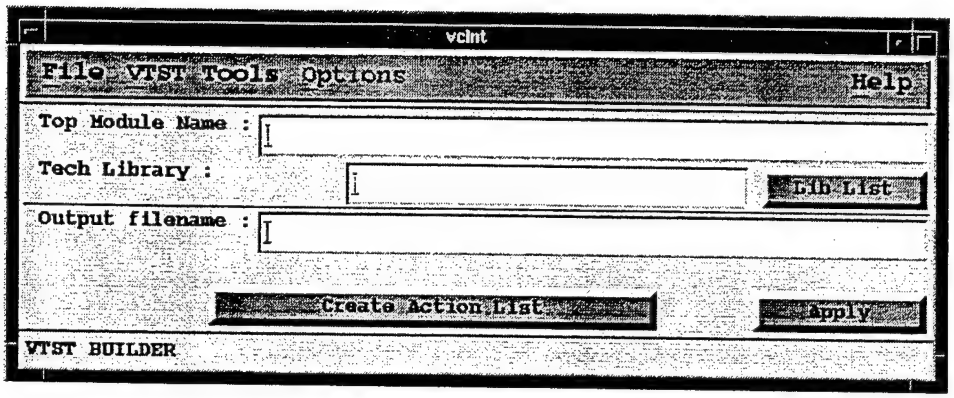


Figure 11.3: User interface for BUILDER

The user can use the VTST client windows interface to invoke BUILDER. To invoke BUILDER windows, the user needs to click on the **VTST BUILDER** on the **Basic** tool list from the **VTST Tools** pull down menu. The BUILDER window will pop up as shown in Figure 11.3.

The user needs to fill in three fields in this window. The first field, **Top Module Name**, is the name of the top most module. The second field, **Tech Library**, is the input technology library name. The third field, **Output Filename**, is the filename for the final BISTed design.

The user can also invoke BUILDER from the command line. To do that, the user needs to type the following information in the command shell.

clnt_builder *0 input_tech_library top_module_name -1 output_filename vtst_host_name current_working_directory*

The first and fourth arguments are fixed to *0* and *-1*. The second argument, *input_tech_library*, is the input tech library name. The third argument, *top_module_name*, is the name of the top most module. The fifth argument, *output_filename*, is the filename for the final BISTed design. The last two arguments, *vtst_host_name* and *current_working_directory*, are used for the distribution system. The *vtst_host_name* is the host name where the *vmgr* is running. The *current_working_directory* is the directory where BUILDER reads and writes related files.

To actually let the BUILDER work, the user needs to set up the following files: **.file**, a file that contains a list of all the NDL netlist files, **.sequence**, a file that contains a list of all the BIST results files, **.inlib**, a file that contains the name of the tech library, and **.ucomplst**, a file that contains all the unknown components' names. Details of these files are explained in the following.

11.3.1 List of NDL Netlist Files

BUILDER needs to know the locations of all the NDL netlist files that are needed for the current design. For any component, if it is not listed on tech library or unknown component list, it must have an NDL netlist file to describe its detailed internal structure. The location of the NDL netlist file (path and filename) needs to be put into a file named *.file* under the current working directory.

11.3.2 List of BIST Action Files

BUILDER needs to know the locations of all the action lists that are related to the current design. The location of the action list (path and filename) needs to be put in a file named *.sequence* under the current working directory.

11.3.3 Input Technology Library

There are two methods to direct the BUILDER which tech library the NDL netlists are using. The user can fill in the information in the *Tech Library* field on **BUILDER Window** (or the *input_tech_library* field on command line mode) with the desired tech library name. The other is to provide the tech library name in *.inlib* file. If no tech library is provided when BUILDER is invoked, BUILDER will read the *.inlib* file to get the tech library.

11.3.4 Unknown Components list

All the unknown components will be listed on a file call *.ucomplst* under the current working directory. From the BUILDER's point of view, when its parser part finds a component on this list, it assigns a special flag on the component's data structure, so that, when the parser is finished, no error message about this component is generated.

Chapter 12

VTST Test Scheme for Automated BIST Design

The BIST architecture described in Chapter 2 is the BIST design generated by VTST. This BIST design is produced primarily for performing BIST on chip with a few control and test signals generated/excited by ATE after fabrication. However, before fabrication, this design must also be tested and verified with respect to its testability and the normal operation of the original CUT in either LSI Logic's C-MDE environment or the commercial CAD/CAE tools that were employed for design and synthesis.

A test scheme is required to perform self-testing simulation on the CAD/CAE environments. This test scheme can also perform self-testing on the ATE environment. For both environments, it is required that the BISTDaTA tools provide the essential and adequate data for the CAD/CAE tools and the ATE to perform self-testing simulations. Furthermore, to understand how to develop the test scheme, let us look at the test scheme done in the C-MDE environment.

Figure 12.1 presents the flow chart of a test scheme of VTST performed in the C-MDE environment. The BISTDaTA tools will generate the BIST design in an NDL netlist and two C-MDE simulation files **.SCL** and **.TPT**. These two files are the simulation control

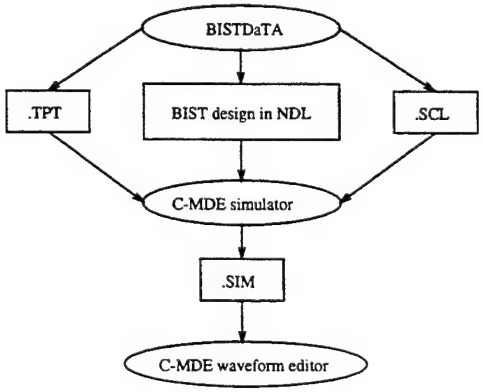


Figure 12.1: VTST simulation interface

file and the simulation pattern file for the C-MDE simulator. C-MDE will read in the BIST design written in NDL and the simulation control and pattern files and the simulator will perform the simulation. From the waveform editor, the *fault_status_flag* will notify the users whether the CUT is fault-free or faulty. If the *fault_status_flag* signal is pulled HIGH, it is fault-free. Otherwise, it is faulty.

In the BIST architecture, the test resource such as the SG module and the RA&C module are shared among CUTs within the chip. For each CUT, the test methodology employed may vary. To effectively test each CUT, users are required to develop a specific test scheme for each CUT and make use of external ATE to perform testing and verification. In general, there are two test schemes: the *non-scan* test scheme and the *scan* test scheme.

In this chapter, we present the basic test schemes for simulating the BIST design produced by VTST. These test schemes can be implemented in other commercial CAD/CAE tool environments for verification of the BIST design with the simulation results done in the BISTDaTA tools. To achieve that, an interfacing file (with extension ".cmde") containing the following data/information must be generated by the BISTDaTA tools.

- initial seed;
- terminating seed;
- the final signature;
- number of test patterns/stimuli to be applied;
- size of test pattern generator;
- size of MISR;
- scan test patterns for scan design.

12.1 Non-Scan Test Scheme

Non-scan test scheme is a test scheme which employs the pattern generation and signature compression method. The control signals used in this test scheme enables the chip performing, terminating self-testing, and validating the chip status (faulty or fault-free). This test scheme is described by the flow chart shown in Figure 12.2. In the following, the relative timing of each signal will be described to perform a non-scan test scheme. And the timing diagram of the critical signals are shown in Figure 12.3)

1. Apply appropriate signals to *Module_Select_Pins* to select the desired CUT.
2. If the CBIST methodology is involved, keep the signal *vtst_sys_xormux_ctrl* HIGH.
3. Keep the signal *vtst_lfsr_load* LOW until loading *seed/signature* process is necessary (see Figure 12.3 where *vtst_lfsr_load* is pulled HIGH to load the seed and signature during the interval Time A and Time B).

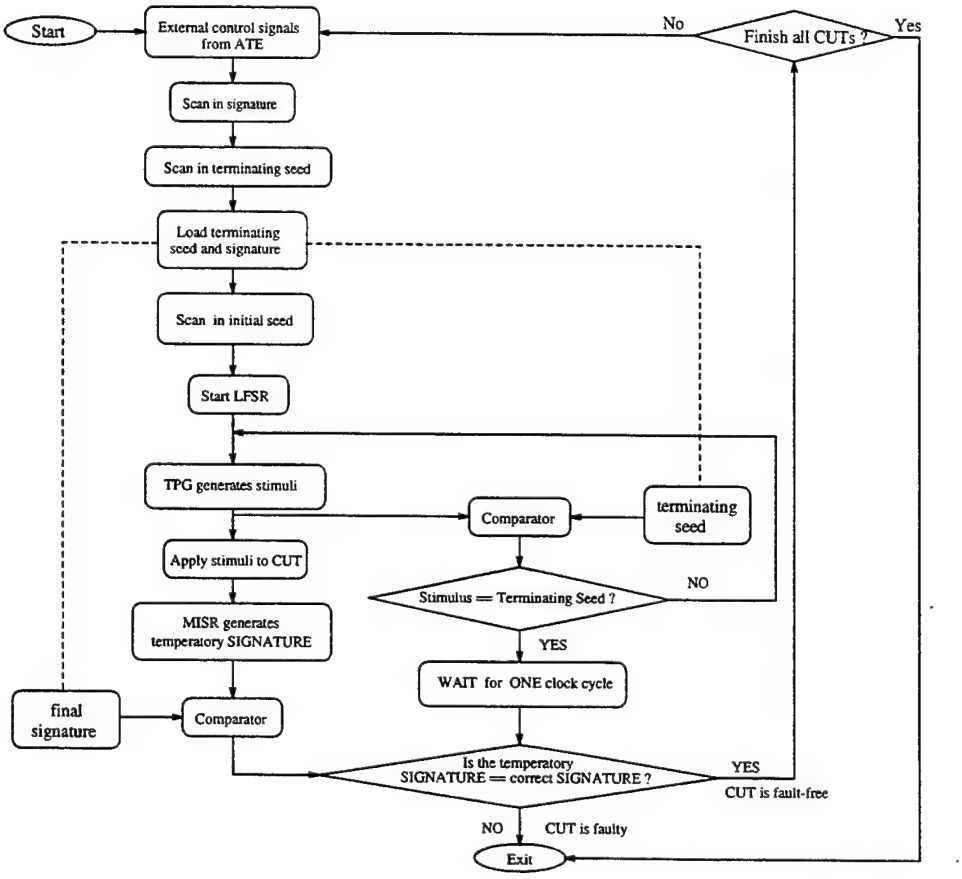


Figure 12.2: System flow for non-scan test scheme

4. Keep the signal *vtst_lfsr_control* LOW for selecting shift mode until LFSR mode is required. As shown in the interval between Time C and Time D in Figure 12.3, the signal *vtst_lfsr_control* is switched from LOW to HIGH to start the LFSR mode in both the TPG and the MISR.

5. Keep the signal *vtst_misr_clr* HIGH unless clearing MISR is desired. As shown in the interval between Time C and Time D in Figure 12.3) the signal *vtst_misr_clr* is pulled LOW to perform the *clear* operation of TPG and MISR when the LFSR mode in both the TPG and the MISR is on and right before the rising edge of the *vtst_sys_clk*.

6. Keep the signal *vtst_misr_set* HIGH unless setting MISR is desired. As shown in the interval between Time C and Time D in Figure 12.3) the signal *vtst_misr_set* is pulled LOW to perform the *set* operation of TPG and MISR when the LFSR mode in both the TPG and the MISR is on and right before the rising edge of the *vtst_sys_clk*.

7. Reset the content of the flip-flops in the TPG for one or two cycles (by pulling *vtst_tpg_clr* LOW for 1 or 2 cycles) to ensure that no data is scanned in while resetting the TPG. This is done only in the very first clock cycles.

8. Set the content of the flip-flops in the TPG for one cycle right after the terminal seed and the signature have been loaded in the TPG and MISR respectively. The signal *vtst_tpg_set* is pulled LOW for 1 cycle right before the *vtst_lfsr_control* is pulled LOW (see Times C and D in Figure 12.3).

9. Keep the signal *vtst_sys_clr* HIGH except that it is desirable to have all 0's in internal non-chained flip-flops when the very first pattern is applied. As shown in the interval between Time C and Time D in Figure 12.3, the signal *vtst_sys_clr* is pulled LOW to *clear* the content of all non-chained flip-flops right before the first pattern is applied.

10. Keep the signal *vtst_sys_set* HIGH except that it is desirable to have all 0's in internal non-chained flip-flops when the very first pattern is applied (see Times C and D in Figure 12.3). As shown in the interval between Time C and Time D in Figure 12.3, the signal *vtst_sys_set* is pulled LOW to *set* all the non-chained flip-flops right before the first pattern is applied.

11. Keep the signal *vtst_mux_ctrl* HIGH all the time because it is in test mode.

12. Scan in the seeds/signature according to the following order with MSB first through the *vtst_lfsr_scan_in* pin.

    - final signature,
    - terminating seed,
    - initial seed.

13. When the last bit of the terminal seed is scanned in, the terminating seed and the final signature should be in the appropriate spot for loading; *vtst_lfsr_load* is then pulled HIGH for one clock cycle for loading. After loading, *vtst_lfsr_load* is kept LOW

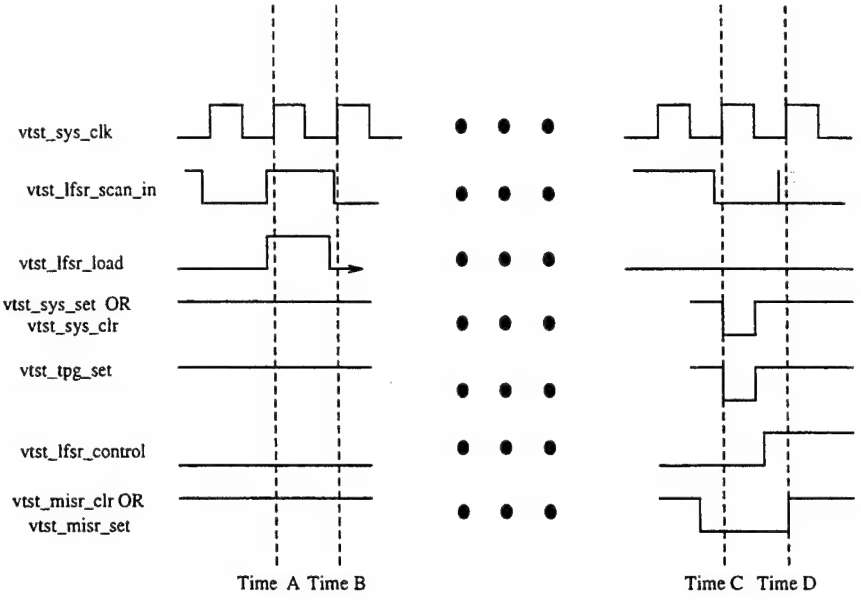


Figure 12.3: Timing diagram for non-scan test scheme

until the next loading (see the interval between Time A and Time B for the signal *vtst_lfsr_load* in Figure 12.3).

14. When the MSB of the initial seed is about to be shifted in, *vtst_misr_set* or *vtst_misr_clr* is pulled LOW for one clock cycle to set or clear the contents of the MISR. After one clock cycle, *vtst_misr_set* or *vtst_misr_clr* should stay HIGH to disable the set/reset operation. When the MSB is shifted into the TPG, *vtst_lfsr_control* is pulled HIGH to enable the TPG/MISR to perform the LFSR mode for generating the next pattern (see Times C and D in Figure 12.3).

When the terminating seed matches the pattern generated by the TPG, the final signature generated by the MISR should match the stored final signature as depicted in Figure 12.2. If this is the case, the fault_status flag should be HIGH to indicate that the CUT is fault-free. Otherwise, it stays LOW, and then the system will test next CUT if there is any.

12.1.1 An Example of Non-Scan Test Scheme

A CUT with 4 inputs and 4 outputs is tested. Assume this 4-input CUT is a BIST design without the CBIST methodology applied. The timing diagram of the non-scan test scan is shown in Figure 12.4. The initial seed 0011, the terminating seed 0010, and the signature is 1110. Since it is the first CUT, the *Module_Select_Pin* stays LOW during testing the first CUT. When the second CUT is tested, then *Module_Select_Pin* is pulled HIGH as shown in Figure 12.4. The TPG is cleared by pulling *vtst_tpg_clr* LOW first and then pulling it HIGH (the same procedure will be done for setting TPG). After it stays LOW, the signature is scanned in first and followed by the terminating seed. When the last bit of the terminating seed is scanned in, the signal *vtst_lfsr_load* is pulled HIGH for one clock cycle to load the seed/signature. After that, the initial seed is scanned in. When the last bit of the initial

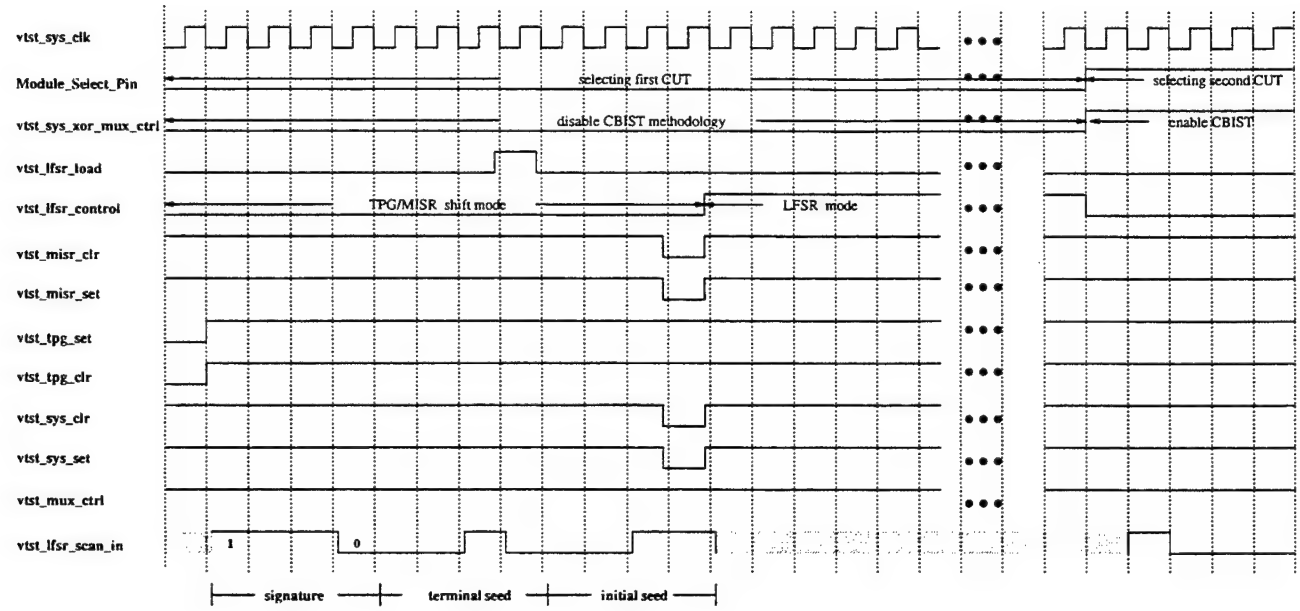


Figure 12.4: Example of timing diagram for non-scan test scheme

seed is scanned in, the signals *vtst_misr_clr* and *vtst_sys_clr* are pulled LOW to clear the the MISR and internal flip-flops. Also at the right edge of the system clock, the signal *vtst_lfsr_control* is pulled HIGH to start the LFSR mode for both the TPG and the MISR.

12.2 Scan Test Scheme

Scan test scheme is a test scheme in which test patterns are scanned in and applied to the CUT and then the MISR in the scan chain will collect the response of each pattern applied and scan out the responses for the verification. Then the scanning and applying is repeated until all the test patterns (deterministic) have been applied and their corresponding responses have been scanned out. For each CUT, there is one individual scan path connecting the TPG to the CUT and then to the MISR. The control signals used in this test scheme enable the chip to perform scanning and applying of individual test pattern. No initial seed, no terminating seed, no final signature and no validating the chip status (faulty or fault-free) are performed on chip. The test scheme is most likely dependent upon the use of ATE. The test scheme can be described by the flow chart as shown in Figure 12.5 and as follows:

1. apply appropriate signals to *Module_Select_Pins* to select the desired CUT.

2. *vtst_sys_scan_enable* is the signal that controls the *enable* port of the scan flip-flop. When the scan pattern is scanned in, the *vtst_sys_scan_enable* signal stays HIGH to select the scan input. When the scan pattern is completely scanned into the desired flip-flops, *vtst_sys_scan_enable* is pulled LOW for one clock cycle to let the scanned pattern be applied to the CUT and then the *vtst_sys_scan_enable* is pulled HIGH again to scan out the responses collected in the internal flip-flops and MISR.

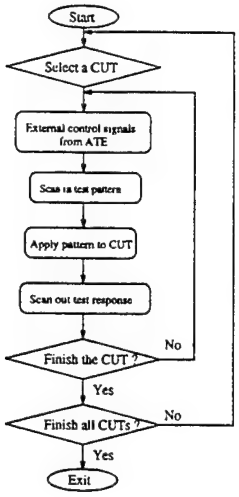


Figure 12.5: System flow for scan test scheme

3. the signal *vtst_lfsr_control* is kept LOW because it is in scan mode. The responses from the CUT will be compressed using XOR gates as long as this signal stays LOW.
4. Keep the signal *vtst_misr_clr* HIGH unless clearing MISR is desired.
5. Keep *vtst_misr_set* HIGH unless setting MISR is desired.
6. Keep *vtst_tpg_clr* HIGH unless clearing TPG is desired.
7. Keep *vtst_sys_set* HIGH unless setting TPG is desirable.
8. Keep *vtst_sys_clr* HIGH except when the very first pattern is applied.
9. Keep *vtst_mux_ctrl* HIGH all the time because it is in test mode.
10. Scan in the test pattern with MSB first through *vtst_lfsr_scan_in* pin.

The following signals will not be used for scan test scheme.

- *vtst_sys_xormux_ctrl*,
- *vtst_lfsr_load*

And they can be anything.

Figure 12.6 depicts the timing diagram for the scan test scheme. At the time A, the last bit of the scanned in pattern will be shifted into the TPG. After that, all patterns should reside in the TPG, the MISR, and all the scan flip-flops. MISR is also cleared. During the interval between time A and B, test results will appear in front of MISR and the chained flip-flops. At the time B, the responses from CUT should be loaded into the MISR and all the flip-flops in the scan-chain. It is the desirable time to pull the signal *vtst_sys_scan_enable* HIGH to load the response to the MISR and flip-flops in scan chain for one clock cycle. At time C, both the MISR and the internal flip-flops in the scan chain should have the responses from the CUT. Therefore, the signal *vtst_sys_scan_enable* is pulled LOW to disable the apply mode, enable the scan mode and scan the responses out.

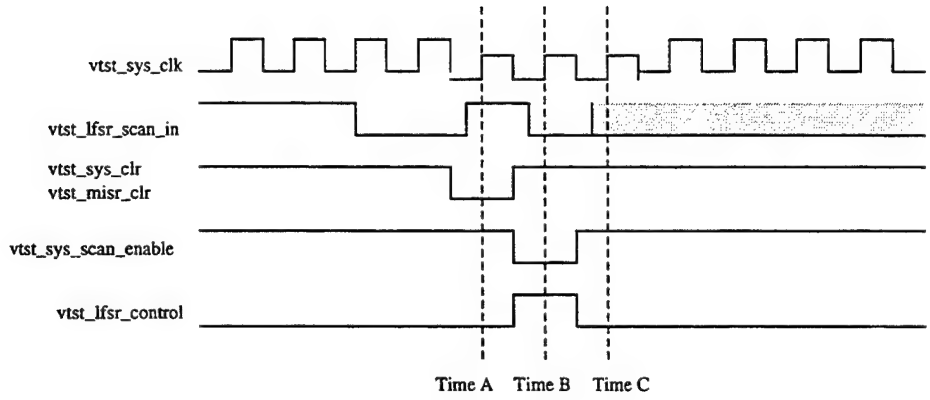


Figure 12.6: Timing diagram for scan test scheme

Chapter 13

VTST Distribution System

With the constraints of area, speed and testability, synthesis of VLSI circuits requires much more computation time and memory space than the traditional VLSI design when the design space is explored. Remote Procedure Call (RPC) in the UNIX operating system provides a way to make use of networked workstation's CPU to enhance or speed up the process of synthesis, design and simulation of VLSI design.

RPC allows a client to execute procedures on other networked computers or servers. The client/server model of computing is a popular model for distributed processing. It applies to any processing environment where one set of entities requests work to be done and another set actually performs the work.

One goal of VTST is to allow the user to explore different synthesis tools at the same time. To avoid all the invoked tools running on one workstation, a RPC-based distribution system has been designed and developed for VTST to achieve the maximum usage of the computer resources on a network, to balance the load among those networked workstations and to achieve the maximum performance of the design.

13.1 Overview

There are four subsystems in the distribution system. They are the VTST tool manager (*vmgr*), VTST tool server (*vtsvc*), VTST tool client (*vclnt*), and VTST synthesis tools. The relations between those four subsystems are shown on Figure 13.1. The user needs to start *vmgr* first, then use the function provided by the *vmgr* to start *vtsvc* on all participating workstations. After starting the *vtsvc* the user can invoke the *vclnt* from *vmgr*. During the initialization phase, the *vtsvc* acts like a client and talks to the server (*vmgr*). After that, it acts as a server and resides on the participating workstation. Inside the *vmgr*, there is a timer which will frequently interrupt the normal operation of *vmgr* and force it to activate a tool server status check routine. During this phase, the *vmgr* acts like a client and talks to the *vtsvc* server on the participating workstations. The *vclnt* always plays a client role; during the initialization phase, it talks to the *vmgr*. When the user requires to dispatch a

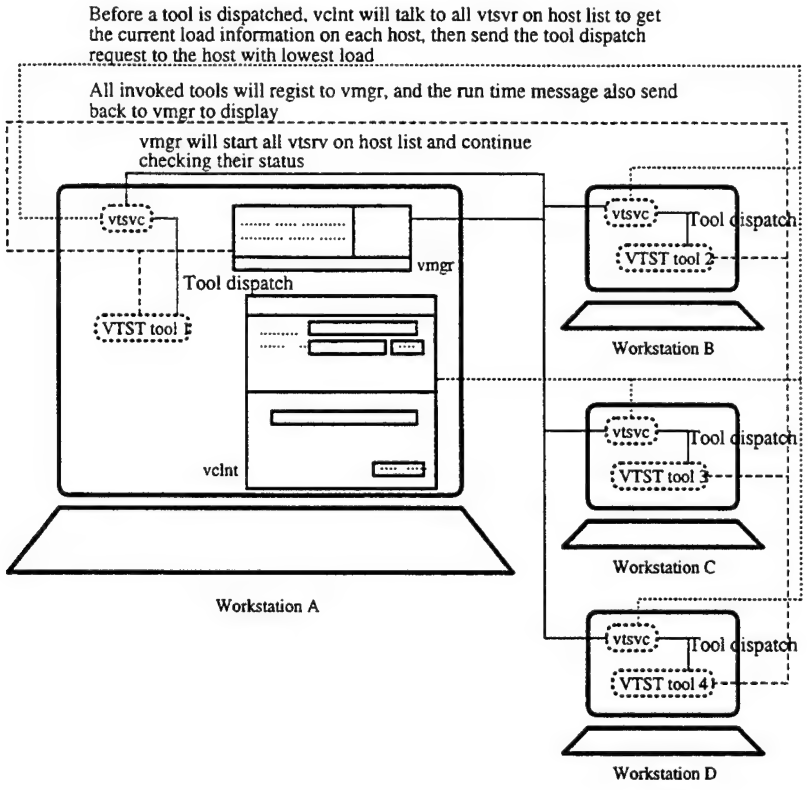


Figure 13.1: VTST and distribution system

synthesis tool, *vclnt* talks to the *vtsvc* server on all participating workstations. A synthesis tool, activated by a *vtsvc* server on a participating workstation under the request sent by *vclnt*, also plays as a client, and can always talk to the *vmgr* server.

13.2 *VMGR* : VTST Tool Manager

VTST tool manager, *vmgr*, is the first subsystem that needs to be invoked in order to start the whole VTST system. It also plays a role as a manager and message server for all other VTST tools. For the manager side, its responsibilities include to start the *vtsvc* on all participating workstations, to start the *vclnt*, to pass the running mode to it, and to check the status of the started *vtsvc* on all networked workstations. For the message server side, it accepts the register request from an invoked tool, and displays a message under the request from a registered tool.

13.2.1 Manage VTST Tool Server.

Before starting the VTST in the distributed mode, the user needs to set up a host list, which is a table containing the name (IP address) of all the participating workstations. After setting up the host list, the user can issue the command to let *vmgr* to start the *vtsvc* on all the participating workstations (refer to Section 14.2 for detailed information on how

	VMGR		VTSVC	
	Role	Function	Role	Function
VTSVC Initialization	Server	Register	Client	Client request Register
Timer Interrupt	Client	Client request server report status.	Server	Response with PID
Other	Server	No Communication	Server	No Communication

Figure 13.2: Client/Server relation between VMGR and VTSVC

to set up this list and start *vtsvc* in distributed mode).

During the *vtsvc* initialization phase, *vtsvc* will behave like a client; it sends a registration request to *vmgr* (server). The registration information includes the process ID, the workstation's name (IP address), and tool ID (it is a predefined number to identify the nature of a tool). When *vmgr* receive a registration request, it will check the tool ID field first. If the ID indicates that the registration request comes from a *vtsvc*, *vmgr* will put this information into a special list to record all started *vtsvc*.

At the same time when the user starts *vtsvc*, *vmgr* will start a timer internally. This timer will interrupt the *vmgr*'s normal function and force it to execute a routine to check the status of the *vtsvc*. During the status checking phase, the *vmgr* will behavior like a client, it will connect to the *vtsvc* server on the host list one by one, and send the request to check status. When a *vtsvc* on a participating workstation receives this request, it will respond to the *vmgr* with its process ID. *VMGR* will restart a *vtsvc* on a networked workstation if the connection fails.

The client/server relation between *vmgr* and *vtsvc* on different time slices are shown on Figure 13.2.

13.2.2 Manage VTST Tool Client.

After *vtsvc* has been started properly, the user can let *vmgr* invoke the *vclnt*. *VMGR* will start the VTST tool client window on the same workstation where *vmgr* is invoked (refer to Section 14.2 for detailed information on how to start the vtst tool client). It, like the *vtsvc* in the initialization phase, will create a client/server relation with *vmgr* and sends a registration request to *vmgr* (server). After initialization, this client/server relation will be broken; no further communication will occur between these two tools.

	VMGR		VCLNT	
	Role	Function	Role	Function
VCLNT Initialization	Server	Register	Client	Client request Register
Other	Server	No Communication	Client	No Communication

Figure 13.3: Client/Server relation between VMGR and VCLNT

The client/server relation between *vmgr* and *vclnt* on different time slots are shown on Figure 13.3.

13.2.3 Register and Deregister VTST Synthesis Tool.

When a synthesis tool is dispatched by *vtsvc*, the dispatched tool will create a client/server relation with *vmgr*. During the initialization, it will send a registration request to *vmgr* first. The register information includes the process ID, the workstation's name, and tool ID. When *vmgr* gets a registration request and identifies it is from a synthesis tool, it will locate a buffer with the information on the registration request to hold run time messages for this tool.

When a synthesis tool is finished, it will send a deregistration request to *vmgr*. It contain the same information as the register request. When *vmgr* gets a deregister request, it will search through the whole buffer area to find the corresponding buffer for this tool, then set up a special flag on this buffer to indicate that the tool is finished.

13.2.4 Display VTST Tool's Run Time Message.

Another main function for *vmgr* is to display a tool's run time message so the user can keep track on a running tool. When a tool needs to display some run time message on the screen, it will send a display message request to *vmgr*. The information in the request includes the message, the process ID, and the workstation's name. When *vmgr* gets a display request for a tool, it will search through the whole buffer area to find the corresponding buffer for this tool, then append the new message to the end of the buffer.

If a tool faces a severe problem such as a segmentation fault or input circuit error, it will send a deregister request with a severe error flag. When *vmgr* gets this kind of request, it will pop up a severe error dialog window to alert the user.

	VMGR		VTST Tool	
	Role	Function	Role	Function
VTST Tool Initialization	Server	Register	Client	Client request Register
VTST Tool Finish	Server	Deregister	Client	Client request Deregister
VTST Tool Error Exit	Server	Deregister Popup up Error Message	Client	Client request Deregister with Severe Error
VTST Tool Running	Server	Display Message	Client	Client Request Display Message
Other	Server	No Communication	Client	No Communication

Figure 13.4: Client/Server relation between VMGR and VTST tool

The client/server relation between *vmgr* and the VTST tool on different time slots are shown on Figure 13.4.

13.3 *VTSVC* : VTST Tool Server.

VTSVC is a background process running on those workstations listed on the host list. It is the core part of the VTST distributed system, because where a tool will be dispatched is controlled by *vtsvc*. The main role of *vtsvc* is a server which accepts requests from both *vmgr* and *vclnt*.

The major functions of *vtsvc* are:

- **Report Status to VMGR.** *VTSVC* is an important unit in the VTST distributed system. One of *vmgr*'s functions is to keep track of all *vtsvc*s, to ensure VTST distributed system's reliability. As we mentioned in Section 13.2.1, *vmgr* will frequently sends the report status request to all *vtsvc*s, to request the updated information. When *vtsvc* receives a request from *vmgr* to report its status, it will response to the request by sending its process ID back to the client (*vmgr*).

- **Report Load to VCLNT.** If the distributed mode is selected, before VTST tool client (*vclnt*) can dispatch a synthesis tool, it will send request for the report of load information to *vtsvc* on all the networked workstations that are on the host list. When the *vtsvc* on a workstation receives this request, it will report the current load back to

	VTSVC		VCLNT	
	Role	**Function**	**Role**	**Function**
User Press Apply Button	**Server**	**Response with System Load**	**Client**	**Client request System Load**
VLCNT Dispatch a Tool	**Server**	**Fork a Child Process to Invoke Required Tool**	**Client**	**Client request Dispatch Tool**

Figure 13.5: Client/Server relation between VCLNT and VTSVC

vclnt. To get the current load of a workstation, *vtsvc* issues a shell command *'w'* by opening an I/O pipe using *popen()*. The UNIX command *'w'* will display a summary of the current activity on the system. The most important part is the heading line which includes the load averages. This load averages means the number of jobs in the run queue averaged over 1, 5 and 15 minutes. The display message will send back to *vtsvc* through an I/O pipe (opened by popen()), and *vtsvc* extracts the number of average jobs in the run queue over 1 minute as the current load.

- **Dispatch a VTST Synthesis Tool.** When *vtsvc* receives a request to dispatch a synthesis tool from *vclnt*, it will invoke another process and the desired tool through the **execlp()** in the newly forked child process. The synthesis tool will create a client/server relation with *vmgr*, and send the run time message and tool's status back to *vmgr*.

13.4 *VCLNT* : VTST Tool Client

VTST tool client, *vclnt*, is the front end user interface that allows the user to set up the working environment and access all the VTST synthesis tools. During the initialization phase, *vclnt* will create a client/server relation with *vmgr* and send a register request to *vmgr* as mentioned in Section 13.2.2.

To invoke a VTST tool, the user just needs to select the proper tool window, fill in the necessary information for the tool, and hit the Apply button. If the distributed mode is selected, *vclnt* will send a report load request to *vtsvc* on all the participating workstations. After *vclnt* receives all the workstation's load information, it will pick up the workstation with the lowest load and send the tool dispatch request to that workstation. The information which *vclnt* sends for tool dispatch include the tool's run time parameters, the host name where the *vmgr* located, and the current work directory. The client/server relation between and *vclnt* and *vtsvc* on different time slots are shown on Figure 13.5.

13.5 VTST Synthesis Tools

When *vtsvc* dispatches a synthesis tool under the request from a *vclnt*, it also pass the information from *vclnt* to the tool. Based on the information, a VTST synthesis tool can connect to the proper *vmgr* (server).

After tool successfully make a connection to *vmgr*, it will register itself to the *vmgr* first. After registration, *vmgr* can accept the display request form this tool. When tool finished, it will send a deregistration request to *vmgr* with a flag which indicates the exit condition.

13.6 VTST on Local Mode

One flexibility of the VTST system is that user can select the local mode instead of the distributed mode. The purpose for VTST to run in local mode is to allows this tool to run on a stand alone system or when the network is not available.

When VTST is running in local mode, all three tools (*vmgr, vtsvc, and vclnt*) are running on the same workstation. All client/server relations and communication remain the same as mentioned in the previous sections. The only difference is when the user invokes a tool, *vclnt* will not request the *vtsvc* to send the current load information, because there is only one workstation involved. For more detailed information of how to set VTST in local mode refer to Section 14.2.1.

13.7 Running VTST on Multiple Workstations

The design goal of the VTST distributed system is not only to distribute jobs to different workstations but also to allow multiple users to use this tool at same time. That is, if user starts the *vmgr* on workstation A, and selects workstation A, B, C, and D as partners, another user can start the *vmgr* on workstation B, C, or D, and also select workstation A, B, C, and D as partners. They both share the same *vtsvc* on participating workstations. To fulfill this goal, the VTST distributed system is designed with the following features:

- *VMGR* can accept any existing *vtsvc* on the participating workstations. If a *vtsvc* is running on a participating workstation, that means another *vmgr* is running on other workstation and selects this workstation as a partner.
- After *vtsvc* is started, it is an independent server. It can accept requests from *vmgr* or *vclnt* on any workstation.
- It is *vclnt*'s responsibility to tell the synthesis tool where the *vmgr* is so that the tool can register and send the display run time message request. A tool, dispatched by *vtsvc* under the request from *vclnt*, will know which *vmgr* it will connect to.

13.8 Summary

The VTST distributed system provides an effective and efficient way to achieve the maximum utilization of the computer resources on a network. Using the distributed system, users can run different test methods concurrently. It can also save a lot of time on the testability synthesis cycle.

Chapter 14

VTST Graphics User Interface

To invoke a VTST synthesis tool, users need to input the necessary parameters. Users also need to remember what is the meaning of each parameter, what are the available options for a parameter, and what is the legal combination between these parameters. Beside these command line parameters, for some synthesis tools, users need to set up run time environment files such as input netlist file, default tech library, etc. On today's computer technology, graphics user interfaces (GUI) are widely used to provide users an easy path to access an application. Users no longer need to remember the meaningless parameters, confused by different options, or spend time in typing to access an application.

Based on these reasons, a Motif based graphics user interface (GUI) for VTST tools has been designed. It is designed to provide user friendly interface to easily invoke all the VTST synthesis tools and set up the run time environments. In this chapter we will talk about how to use this interface.

14.1 Overview

There are two parts in the GUI system, one is the VTST tool manager window (*vmgr*), the other is the VTST tool client window (*vclnt*). The function for the vmgr is to provide a path for users to start the VTST system and display the tool's run time message. The function for the vclnt is to provide a path for users to access VTST tools, which includes to set up the run-time environments, to set up the tool's parameters and options and to access tools.

To invoke the GUI for VTST properly, users need to follow the following steps :

- Start *vmgr*.
- Start VTST tool server.
- Start *vclnt*.

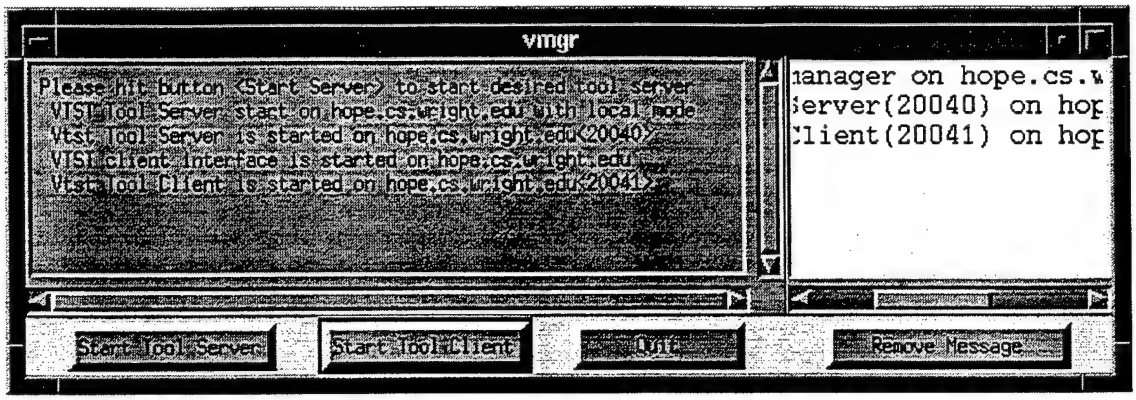


Figure 14.1: VTST tool manager (VMGR)

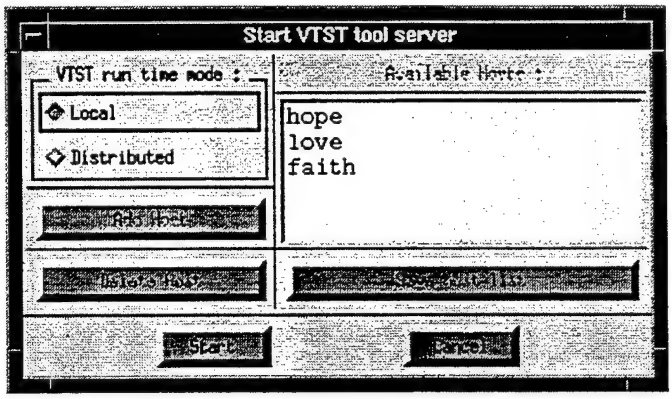


Figure 14.2: Start local tool server

14.2 Using VMGR

To run VTST with GUI , the users must invoke the VTST tool manager first. VTST GUI is built for X-window under Motif window manager, users must use a X-window based terminal and with X-window server running on it. To start the tool server, the user just types

vmgr

under a command shell. A VTST tool manager window will pop up as shown in Figure 14.1. Basically, there are three areas on that window, the upper-left part is the message window, the upper-right part is the registered VTST tools list window, the bottom part is the command area.

14.2.1 How to Start VTSVC

To start the VTST tool server, the users need to press the Start Tool Server button. A *Start VTST tool server* dialog window shown as Figure 14.2 will pop up. As we mentioned in the beginning that VTST can be in local or distributed mode. The local mode means

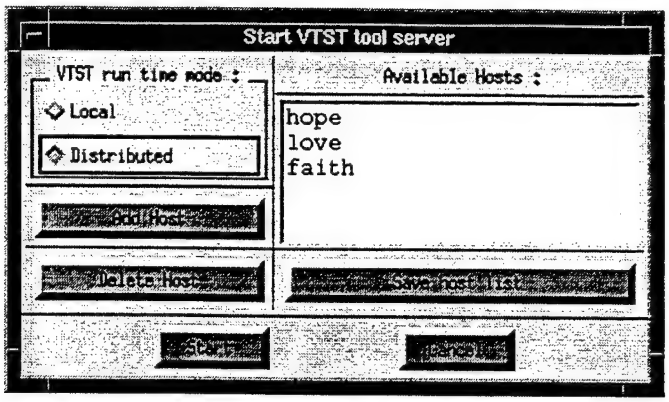


Figure 14.3: Start distributed tool server

Figure 14.4: Add new host

that only one *vtsvc* is started and that one resides on the same workstation as where the *vmgr* and *vclnt* are running. The distributed mode means the *vtsvc* will be started on all networked workstations.

- **Setup Local Mode**

  To set VTST to local mode, click Local inside the *VTST run time mode* toggle box on *Start VTST tool server* dialog window.

- **Setup Distributed Mode**

  To set VTST to distributed mode, click distributed inside the *VTST run time mode* toggle box on *Start VTST tool server* dialog window. The dialog window will enable those parts for setting up networked workstations. The new dialog window is shown in Figure 14.3. *VMGR* will read a file, **.vtsthost**, under the same directory where it is invoked to get the list of networked workstations. The user can use this dialog window to add or delete a networked workstation on this list.

  To remove a networked workstation from this list, the user just highlights the workstation in the *Available Host* list box, then press the Delete Host button.

  To add a new workstation to this list, the user just presses the Add Host button. Another window *New host* will pop up on the screen as shown in Figure 14.4. The user can type in the new host name or IP address in the text input window, then press the Add button. After all the new names have been added, the user can press the Done button to close this pop up window.

After modifying the host list, the user **must** press the Save host list button to save the modified host list to the **.vtsthost** file.

After settings up the VTST run time mode and the networked workstations (if distributed mode is select), users just need to press the Start to start the *vtsvc* on desired workstations.

14.2.2 How to Start VCLNT

The user can start *vclnt* by pressing the Start Tool Client button. The user must start *vtsvc* first in order to invoke *vclnt*.

14.2.3 How to View a Tool's Run Time Message

To view a particular tool's run time message, the users need to double click that tool on the tool list window. The corresponding run time message will be displayed on the message window.

14.2.4 How to Remove a Tool's Run Time Message

To remove a particular tool's run time message, users need to highlight that tool on the tool list window, and then hit the Remove Message button. The highlighted tool will be removed from the list, so as its message.

14.2.5 How to Quit

The only trouble free way to quit VTST tools is to hit the Quit button on *vmgr*. It will kill all VTST related processes on all networked workstations.

14.3 Using VCLNT : Environment Set Up

All the environment related setup dialog windows can be invoked from the submenu under System File .. under File pull down menu. Figure 14.5 shows the submenu under *System File*.

14.3.1 Set Current Working Directory

Working directory is the directory where all VTST tools' file I/O will take place. VTST defaults to the directory where *vmgr* is invoked. The working directory can be changed by

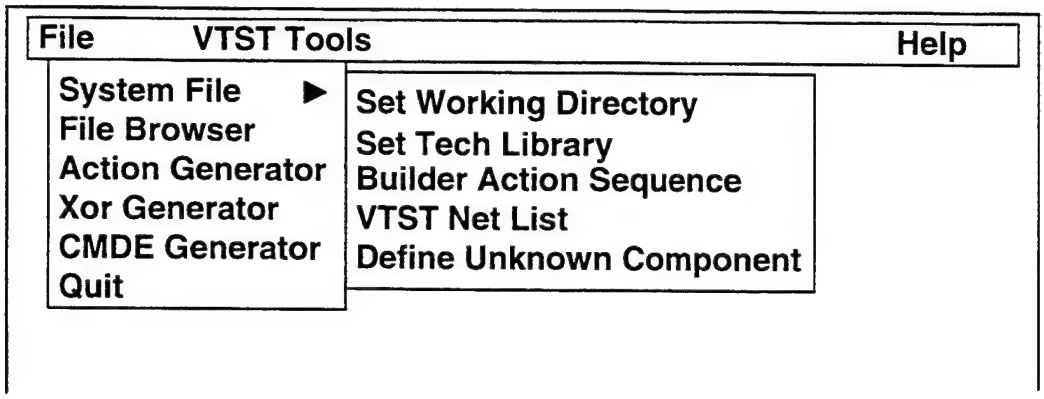


Figure 14.5: VCLNT system file pull down menu

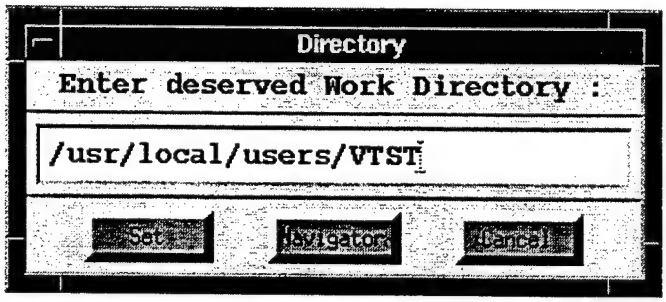


Figure 14.6: VCLNT: set working directory

using the *set working directory* dialog window shown as Figure 14.6. To invoke the window, click

File ⟶ System File .. ⟶ Set Working Directory

The default directory is shown on the text input field. The user can change the working directory by typing the new directory in, or using the directory navigator to browse the file system and navigate to the desired working directory. To set up the new working directory, press the Set.

To invoke the directory navigator, press the Navigator button on the *set working directory* window, the navigator will pop up as shown in Figure 14.7. Users can navigate through the file system by double clicking the directory under the directory list window (the left one of the two file list windows). After navigating to the desired directory, press the Done button. The new directory will pop up the text input field on the *set working directory* window.

14.3.2 Set Tech Library

VCLNT will read in three files, *.libpath*, *.inlib* and *.outlib*, under current working directory to get tech library related setting which includes the tech library path and the default input and output libraries. Users can update those information by clicking

File ⟶ System File .. ⟶ Set Tech Library

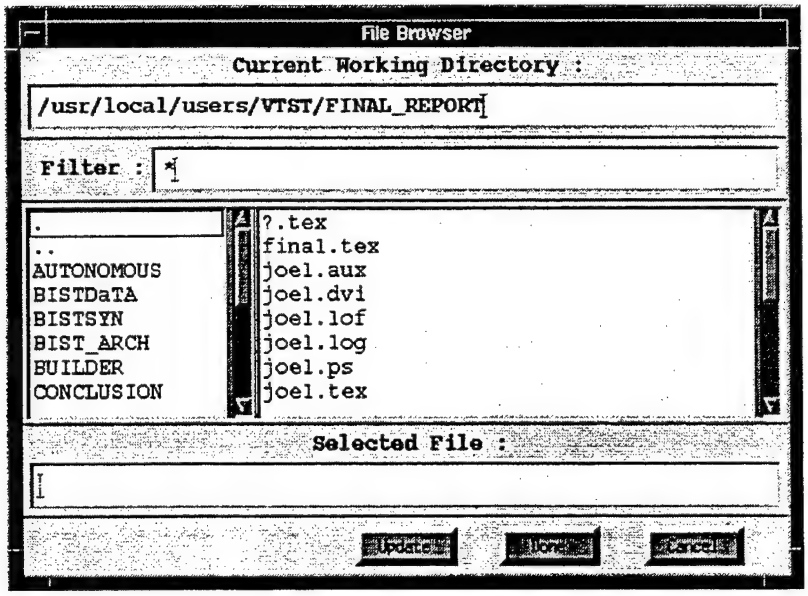


Figure 14.7: Directory navigator window

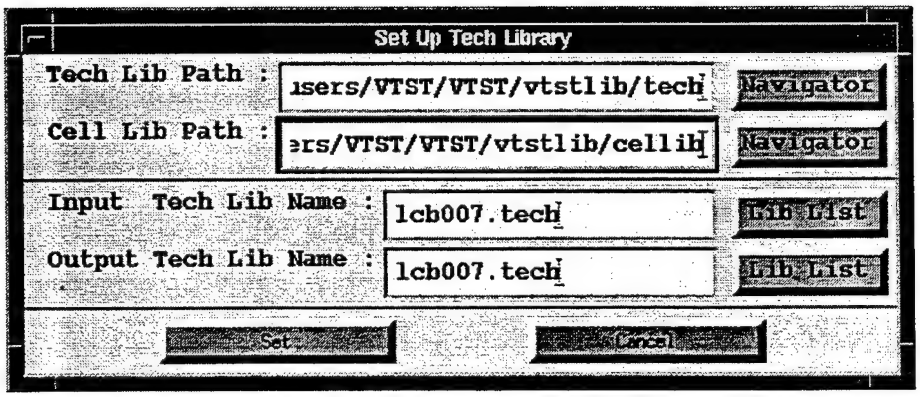


Figure 14.8: VCLNT: set technology library

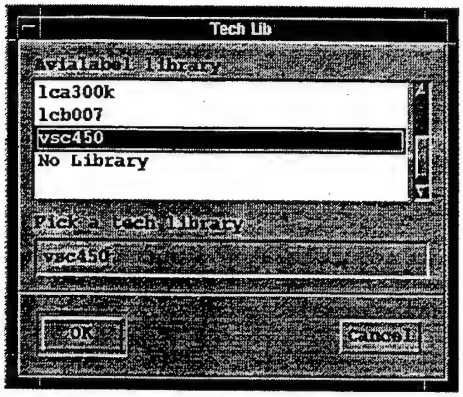


Figure 14.9: Technology library list

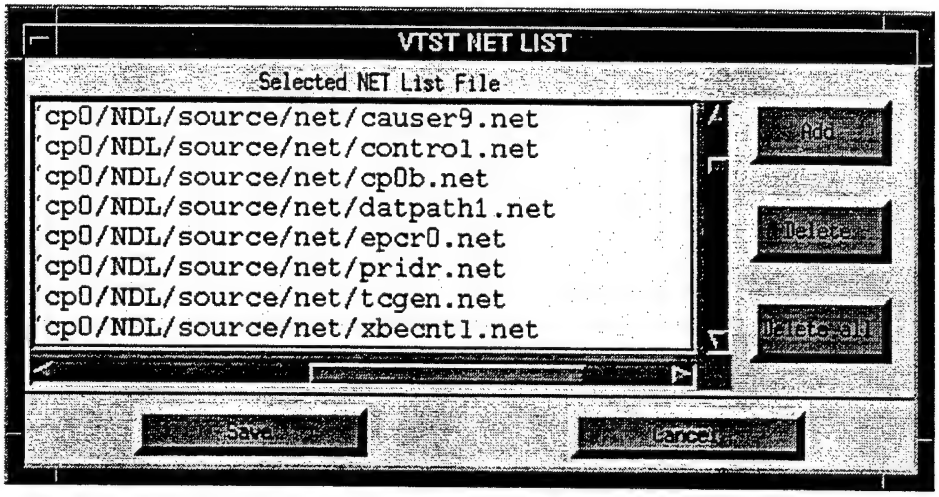


Figure 14.10: VCLNT: set netlist

the *Set Tech Library* dialog window will pop up as shown in Figure 14.8. Using this dialog window, the *Tech Lib path*, *Cell Lib Path*, *default input tech library*, and *default output tech library* can be set up. To set up those paths, users can either type the new path in or invoke the directory navigator as mentioned in the previous section. To set up the default input (output) tech library, users can either type the new library name in or press the Lib List button to get a list of available tech library names as shown in Figure 14.9, then select the desired library from the list.

After filling in all the information, press Set button to save the new information in those files as we mentioned at the beginning of this Section.

14.3.3 Set Net List File

A design input to the VTST system is most likely a hierarchical design. It is unlikely that all modules/components in the design are written in a single file. They will be scattered in different directories and filenames. VTST allows users to include all files regarding the design to reside in different directories/filenames. The net list file, or known as *.file*, is a file

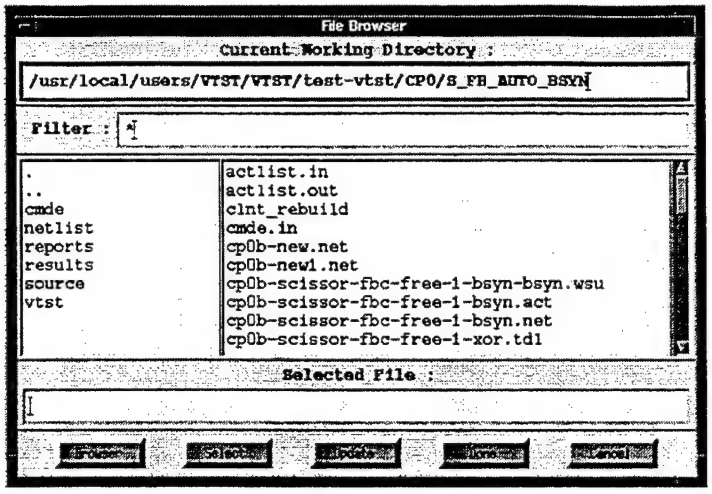


Figure 14.11: Multiple file selection dialog window

that stores the full path and filenames of all the files that describe the design. The VTST parser and BUILDER tools will read this file to figure out where are the files. *VCLNT* provide a dialog window for users to set up this file. To invoke the *Set Net List File* dialog window, click

File ⟶ System File .. ⟶ VTST Net List

the dialog window will pop up as shown in Figure 14.10. Basically, there are two parts in this window: one is a file list part which contains all the selected files; the other is the command part where users can press the button in this area to modify the file list.

To add new files to this list, press the Add button, a *multiple file selection* dialog window will pop up as shown in Figure 14.11. The same navigation function as mentioned in the *directory navigator* in Section 14.3.1 also provided in this dialog window, to allow users to navigate to the desired directory. After navigating to the desired directory, users can select the desired file under this directory by clicking the file listed on the file list window (the right part of the two file list windows). The selected file will appear on the *selected file* section, at this point, users can press the Select button, the desired file will be added to the file list part in the *Set Net List File* dialog window. Continuing the same steps, users can select all the desired files through the *multiple file selection* dialog window. When finish the file adding, press the Done button to close the dialog window.

To delete a file from the file list, just highlight the file and press the Delete button. To delete all files, just press the Delete All button.

After all the desired files are in the list, press the Save button to save the new file list to the *.file* file under the current working directory.

14.3.4 Set Action Sequence

The action sequence file, *.sequence*, is used to tell BUILDER where are those action lists and what is the order of those files. To setup action sequence file, click

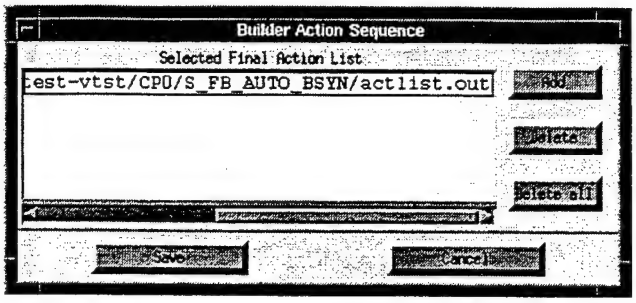


Figure 14.12: VCLNT: set BUILDER action sequence

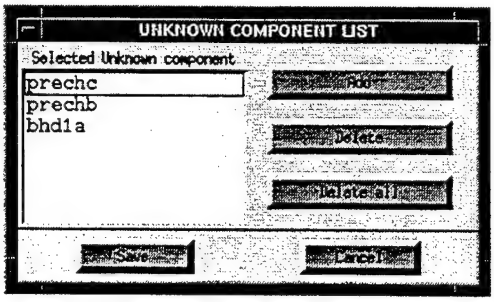


Figure 14.13: VCLNT: set unknown component

File ⟶ System File .. ⟶ BUILDER Action Sequence

The *Set BUILDER Action Sequence* dialog windows will pop up as shown in Figure 14.12. This window has the same orientation as the *Set Tech Library* dialog window (refer to Section 14.3.3 for how to modify the list).

After all the desired action list files have been selected and placed in the correct order, press the Save button to save the new file list to the *.sequence* file under the current working directory.

14.3.5 Set Unknown Component List

It is unlikely that all the complex components will be described in detail in the technology library mapping file. For those complex or not easily defined components such as ROM, RAM, it is more effective to isolate them from the circuit under test and test them separately. To do that, VTST tools will bypass them if users list them in the *.ucomplst* file under the current working directory. To setup this list from the window, click

File ⟶ System File .. ⟶ Define Unknown Component

the *unknown component list* dialog window will pop up as shown in Figure 14.13. To add a new unknown component to this list, users just press the Add Host button. Another window *new unknown component* will pop up on screen as shown in Figure 14.14. Users can type in the new unknown component in the text input window, then press the Add button. After all the new names have been added, users can press the Done button to close this pop up window (refer to Section 14.3.3 for how to remove a item from the list).

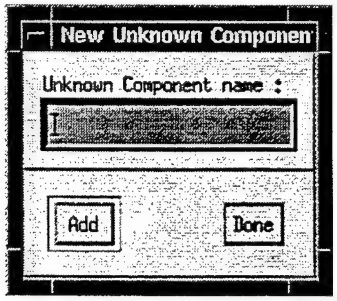


Figure 14.14: VCLNT: add new unknown component

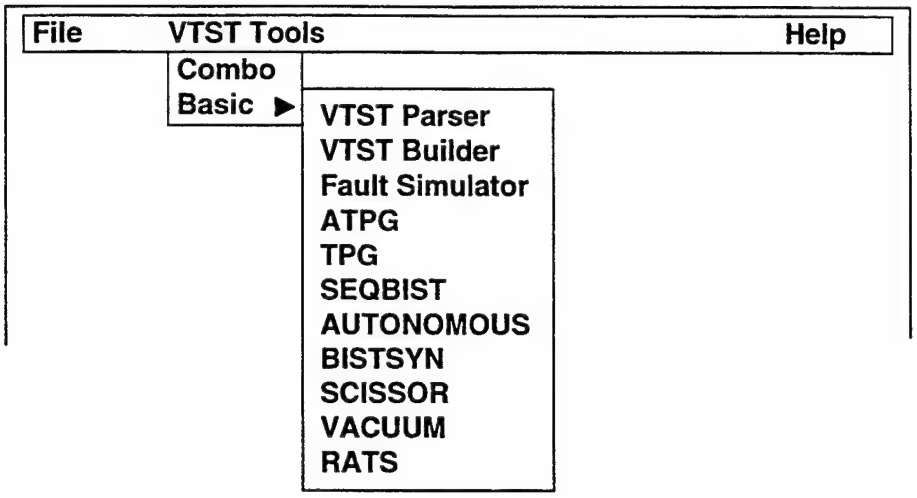


Figure 14.15: VCLNT tool menu

After all the unknown components are added to the list, press the Save button to save the new list to the *.ucomplst* file under the current working directory.

14.4 Using VCLNT : Tool Invocation

Most of the VTST synthesis tools can be invoked from the submenu under Basic under VTST Tools pull down menu. Figure 14.15 shows the submenu under *Basic*. Other synthesis related tools such as action, XOR, and CMDE generator, can be invoked from the *File* pull down menu (Figure 14.5) or from related synthesis tool.

There are some common features on all the synthesis tool windows :

- **Window Orientation.** Basically, the window can be divided into two parts: the first part is used for users to set the necessary parameters and options for the tool to run, and the second part is used to tell the tool how to write reports. The Apply button is located on the bottom-left corner of the window. If there is tool related utility, the invocation button will be located on the left side next to the Apply button.

- **Input File Selector.** For all the file related text input fields, there is a Select

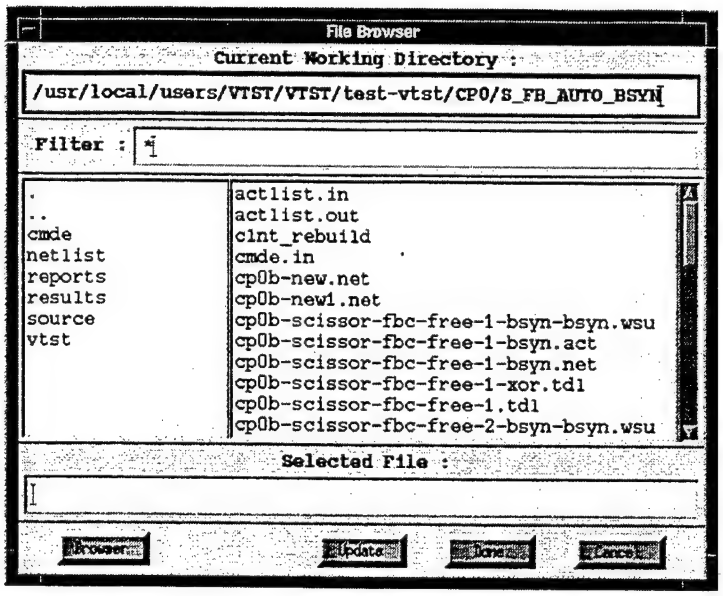


Figure 14.16: Single file selection dialog window

button next to the right edge of the text input field. This button can invoke a *Single File Selection* dialog window shown in Figure 14.16. This window looks the same as the *Multi File Select* dialog window but does not have the Select button. Users can use this window to select the desired file as mentioned in Section 14.3.3. When the desired file is selected, press the Done. Then, the file will appear in the related text input field.

- **Tech Library List.** For all the tech library related text input field, there is a Lib List button next to the right edge of the text input field. This button can invoke the *Tech Library List* as shown in Figure 14.9, users can select the desired library form this list.

Users must set the library path environment file under the current working directory, it provides library path for VTST tools. If the desired tech library is not located under the current directory, users must set up this file (refer to the Section 14.3.2 for how to set up default tech lib path).

14.4.1 Parser

VTST parser is a tool used to translate a net description language from one format to another format. Currently, the VTST parser can provide the translation among **NDL**, **TDN** and **VHDL**. To invoke the parser window, click

VTST Tools $\longrightarrow$ Basic $\longrightarrow$ VTST Parser

the *vclnt* window will change to parser mode as shown in Figure 14.17. The followings are the explanation for each field on this window.

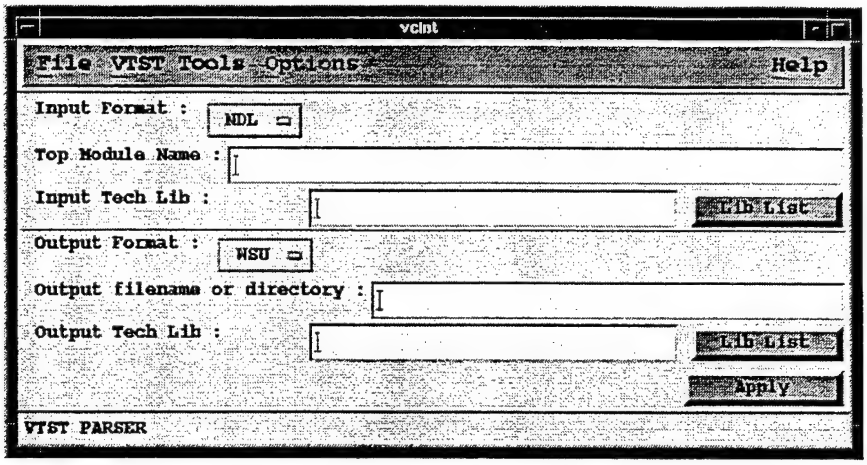


Figure 14.17: VTST Parser

- **Input Format :** This field tells the parser which net description language of the circuits is used. Users can use this option menu to select the input language.
- **Top Module Name :** This field tells the parser which module is the top module. Parser only translates the top module and all the modules used under the top module to the new format.
- **Input Tech Lib :** This field tells the parser which tech library of the input circuits is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let the parser use the default input library (*.inlib*)).
- **Output Format :** This field tells the parser which net description language of the input circuit is translated to. Users can use this option menu to select the output net description language.
- **Output Filename or Directory :** This field tells the parser where the translated circuits will be written to. If the output language is VHDL, parser will treat this field as a directory, all the translated VHDL files will be written under this directory.
- **Output Tech Lib :** This field tells the parser which tech library of the translated circuits is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default output library (*.outlib*)).

Users also need to set up the following related files under current working directory to let parser work correctly :

- **Input Net List File (.file) :** The VTST parser needs to know where are the files that describe the design. This information is provided in the *.file* file (refer to the Section 14.3.3 for how to set up the *.file* file).
- **Default Tech Library (.inlib and .outlib) and Tech Path (.libpath) :** If users do not provide any information about the input/output tech library (leave those field

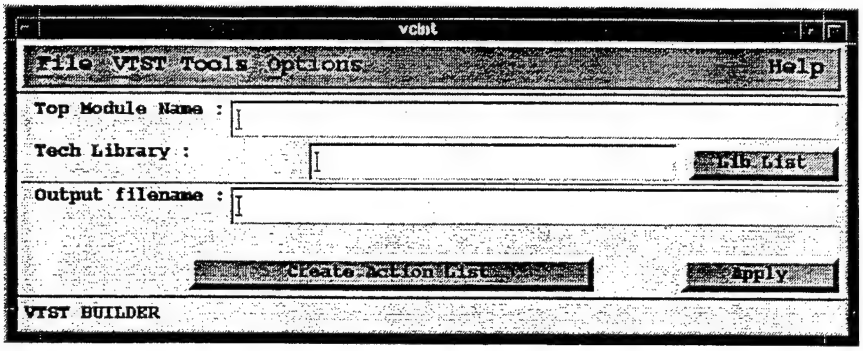


Figure 14.18: VTST BUILDER

blank), the parser will look up *.inlib* and *.outlib* files to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in current working directory (refer to the Section 14.3.2 for how to set up *.inlib*, *.outlib*, and *.libpath* files).

- **Unknown Component List (.ucomplst) :** The VTST parser needs to know which components are unknown so parser can bypass those components. This information is provided in *.ucomplst* file (refer to the Section 14.3.5 for how to set up *.ucomplst* file).

14.4.2 BUILDER

VTST BUILDER is a tool used to automatically insert test points and components into the circuit. Currently, BUILDER can only accept NDL netlist as the input net description language. To invoke the BUILDER window, click

VTST Tools ⟶ Basic ⟶ VTST BUILDER

the *vclnt* window will change to BUILDER mode as shown in Figure 14.18.

The following is the explanation for each field on this window :

- **Top Module Name :** This field tells the BUILDER which module is the top module. BUILDER only modifies the circuit under the top module.
- **Tech Lib :** This field tells the parser which tech library of the input circuits is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).
- **Output Filename :** This field tells the BUILDER where the BISTed circuits will be written to.

Users also need to set up the following related files under current working directory to let BUILDER work correctly :

- **Input Net List File (.file) :** VTST BUILDER needs to know where are the files that describe the design. This information is provided in the *.file* file (refer to the Section 14.3.3 for how to set up the *.file* file).

- **BUILDER Action Sequence File (.sequence) :** VTST BUILDER needs to know where are those action list files. This information is provided in the *.sequence* file (refer to the Section 14.3.4 for how to set up the *.sequence* file).

- **Library path (.libpath) :** It is the library path environment file, it provides the library path for VTST tools. If the desired tech library does not locate under current directory, users must set up this file (refer to the Section 14.3.2 for how to set up default tech lib path).

- **Default Input Tech Library (.inlib) and Tech Path (.libpath) :** If users do not provide any information about the input tech library (leave those field blank), BUILDER will look up *.inlib* file to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in the current working directory (refer to the Section 14.3.2 for how to set up *.inlib* and *.libpath* file).

- **Unknown Component List (.ucomplst) :** VTST BUILDER needs to know which components are unknown so parser routine can bypass those components. This information is provided in *.ucomplst* file (refer to the Section 14.3.5 for how to set up *.ucomplst* file).

14.4.3 Fault Simulator

VTST fault simulator (fsim) is the tool used to analyze a circuit's stuck-at faults. To invoke the fsim window, click

VTST Tools ⟶ Basic ⟶ Fault Simulator

the *vclnt* window will change to fsim mode as shown in Figure 14.19.

The followings are the explanation for each field on this window:

- **Circuit File :** This field tells fsim what is the circuit that fsim will work on.

- **Tech Lib :** This field tells the fsim which tech library that the input circuit is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).

- **Internal DFF Status Option :** This option menu tells fsim how to set the initial state of DFFs inside the test circuit. The available options are *RESET* (set initial state to 0), *SET* (set initial state to 1), and *Don't Care* (do not care the initial state).

- **Pattern Mode Option :** This option menu tells fsim how to generate the test pattern. There are four options *Random Generate*, *From Pattern File*, *From LFSR*,

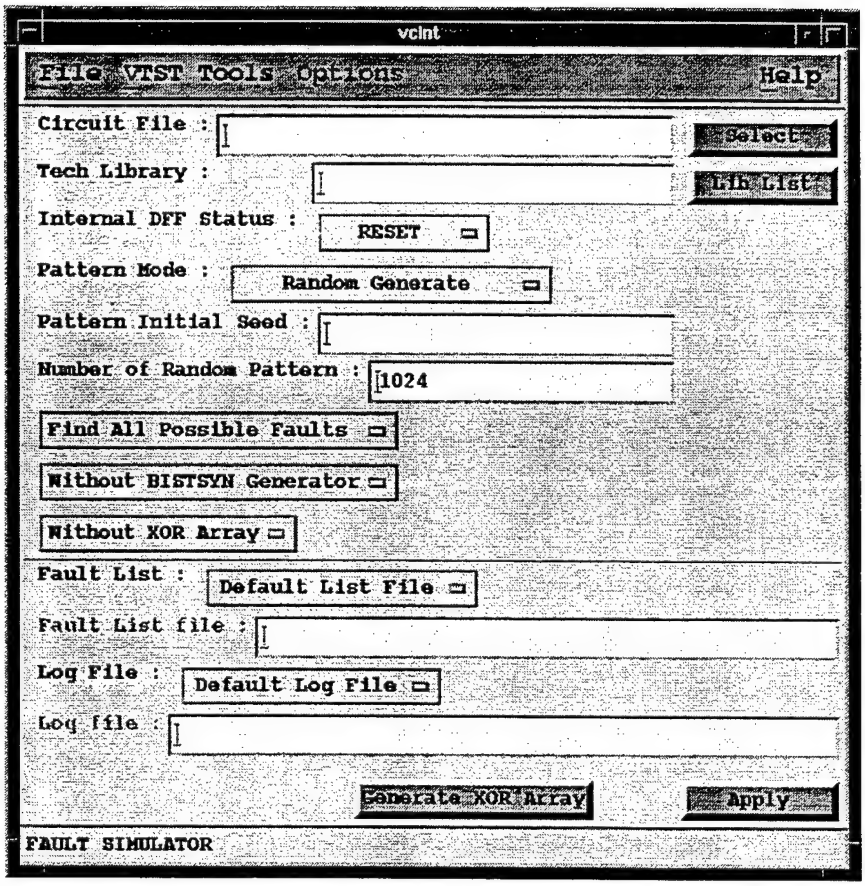


Figure 14.19: VTST fault simulator

and *From Cellular Automata*. *Random Generate* means the test patterns are generated by a random generator built inside the fault simulator. *From Pattern File* means the test patterns are read in from an external pattern file. *From LFSR* means the test patterns are generated according to linear feedback shift register architecture. *From Cellular Automata* means the test patterns are generated according to cellular automata architecture.

- **Pattern Initial Seed :** If the pattern generation mode is not selected from *From Pattern File*, user needs to set the initial seed field. The initial seed has two meanings depending on the selected pattern generation mode. If *Random Generate* mode is selected, this field means the initial seed for the internal random generator. If *From LFSR* or *From Cellular Automata* is selected, this field means the first test pattern.

- **Number of Pattern / Test Pattern File :** If the pattern generation mode is not selected as *From Pattern File*, this field is used to tell fsim how many patterns fault simulator needs to simulate. If the *From Pattern File* mode is selected, a text input box will appear on the right side next to this option menu, users need to fill the pattern file in.

- **Fault Option :** This option menu tells fsim what faults fsim will work on. Users can select *Find All Possible Faults* to let fsim find the faults as many as possible based on test patterns. On the other hand, users can select *Specified Faults* by telling the fsim to concentrate on a set of predefined faults to see how good is the current patterns for this set of faults. If the second mode is selected, a text input box will appear on the right side next to this option menu, users need to fill in the fault list file.

- **BISTSYN Generator Option :** This option menu tells fsim if BISTSYN generator is used. BISTSYN generator is a piece of circuit generated by BISTSYN tool. It is used to reduce the size of LFSR. If BISTSYN generator is selected, users need to assign the BISTSYN generator file in the text box appear on the right side next to this option menu.

- **XOR Generator Option :** This option menu tells fsim the location of XOR generator if it s selected. Users must use the XOR generator if the output size is greater than 64. Users can assign the XOR generator file in the text box which appears on the right side next to this option menu.

- **Default Input Tech Library (.inlib) and Tech Path (.libpath) :** If users do not provide any information about the input tech library (leave those field blank), fsim will look up *.inlib* file to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in current working directory (refer to the Section 14.3.2 for how to set up *.inlib* and *.libpath* file).

The fsim window is shown in Figure 14.20.

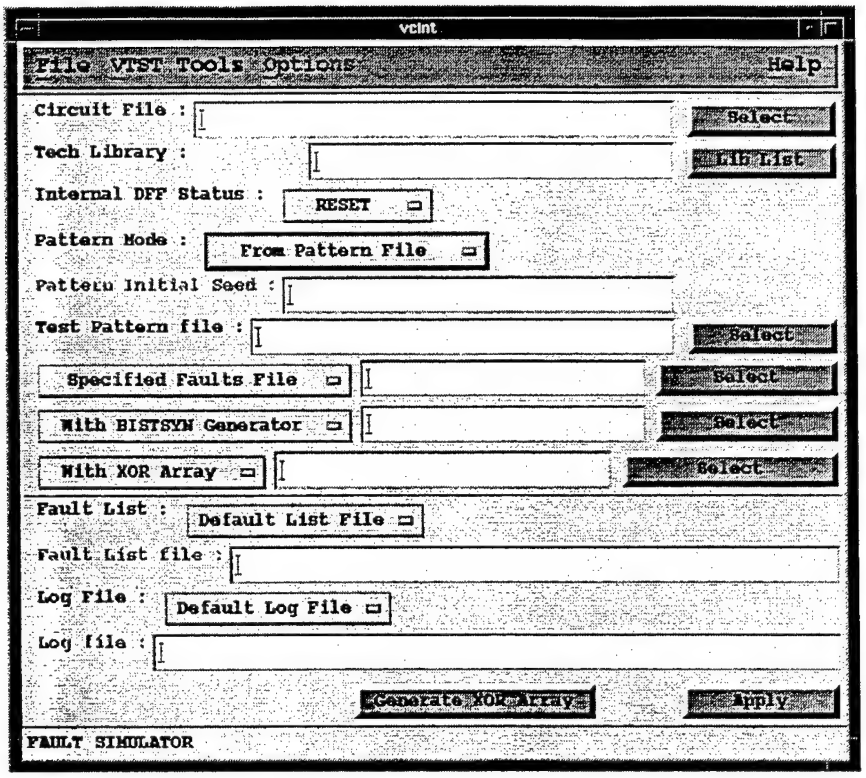


Figure 14.20: VTST fault simulator

14.4.4 ATPG

VTST Automatic Test Pattern Generator (ATPG) is the tool used to analyze a combinational circuit and find the test patterns and the associated fault coverage. To invoke the ATPG window, click

VTST Tools ⟶ Basic ⟶ ATPG

the *vclnt* window will change to ATPG mode as shown in Figure 14.21.

The following is the explanation for each field on this window:

- **Circuit File :** This field tells ATPG what is the circuit that ATPG will work on.
- **Tech Lib :** This field tells the ATPG which tech library of the input circuit is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).
- **Number of Backtracks :** This field specifies the number of backtracks that will be performed in the ATPG program.
- **ATPG Mode Option :** This option menu provides users the option of performing ATPG based on either (1) all the faults present in the circuit, or (2) the specified faults. Users can select *Find All Possible Faults* and find the fault coverage. On the other hand, users can select *Specified Faults* and find the fault coverage.

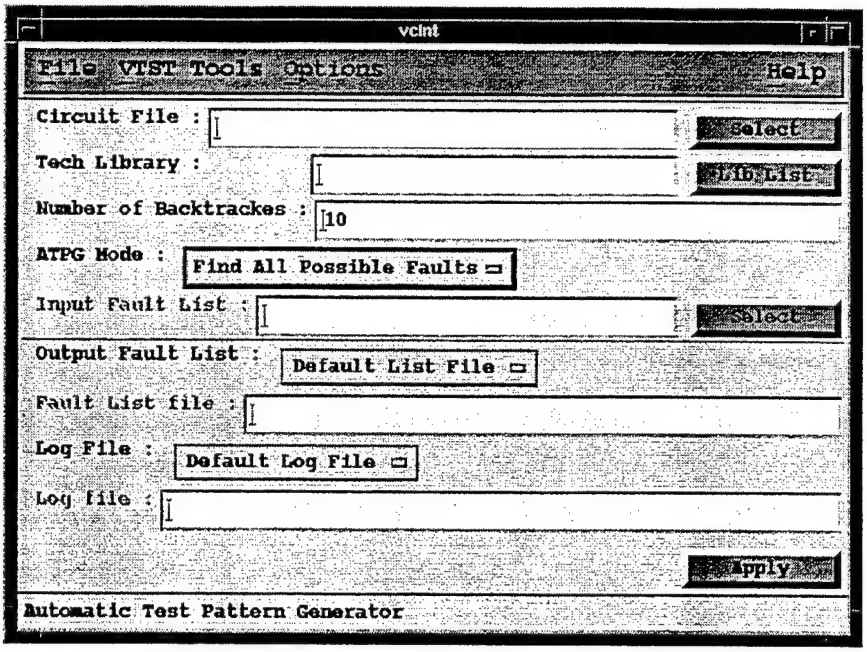


Figure 14.21: VTST automatic test pattern generator

- **Input Fault List :** This field is used only if users select *Specified Faults* as ATPG's mode. It tells ATPG where is the file that contains the specified faults.

Users also need to set up the following related files under current working directory to let ATPG work correctly :

- **Default Input Tech Library (.inlib) and Tech Path (.libpath) :** If users do not provide any information about the input tech library (leave those field blank), ATPG will look up *.inlib* file to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in current working directory (refer to the Section 14.3.2 for how to set up *.inlib* and *.libpath* file).

14.4.5 TPG

VTST Test Pattern Generator (TPG) is the tool that uses the linear feedback shift register (LFSR) architecture to generate the test patterns. To invoke the TPG window, click

VTST Tools ⟶ Basic ⟶ TPG

the *vclnt* window will change to TPG mode as shown in Figure 14.22.

The following is the explanation for each field on this window :

- **Number of Bits:** This field specifies size of each test pattern.
- **Number of Patterns :** This field specifies number of patterns that TPG will generate.

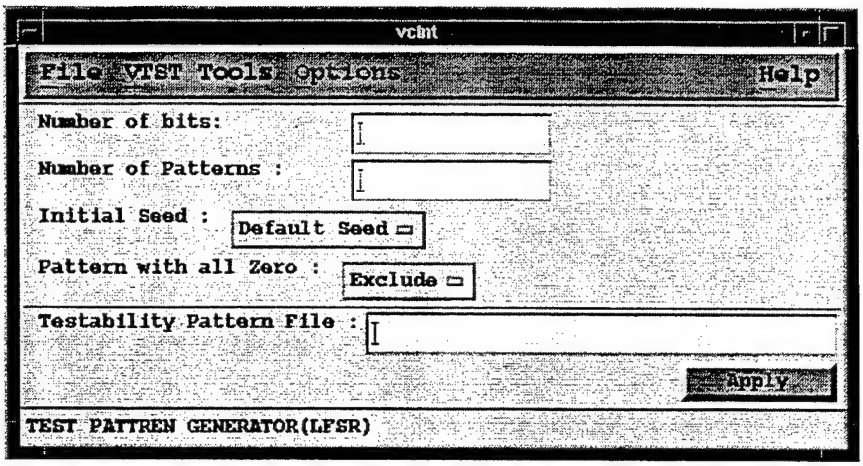


Figure 14.22: VTST test pattern generator

- **Initial Seed :** This option menu allows users to select the rule to generate initial seed. Users can select *Default Seed* or *Random Seed.* If users select *Default Seed*, then TPG will create a seed with all bits '0' except the LSB. The LSB will be set to '1'. On the other hand, if *Random Seed* is selected TPG will use random generator to generate the seed.

- **Pattern with All Zero :** This option menu allow users to decide to include all zero pattern or not.

- **Testability Pattern File :** This field specifies the file name where TPG will write the patterns to.

14.4.6 SEQBIST

VTST SEQBIST tool is used to analyze the testability of a sequential circuit. To invoke the SEQBIST window, click

VTST Tools ⟶ Basic ⟶ SEQBIST

the *vclnt* window will change to SEQBIST mode as shown in Figure 14.23.

The following is the explanation for each field on this window :

- **Circuit File :** This field tells SEQBIST what is the circuit that SEQBIST will work on.

- **Tech Lib :** This field tells the SEQBIST which tech library of the input circuit is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let the parser to use the default input library (*.inlib*)).

- **SEQBIST Mode :** Users can use this option menu to specify the desired SEQBIST mode. There are four different mode in SEQBIST, they are *Circular BIST (CBIST)*,

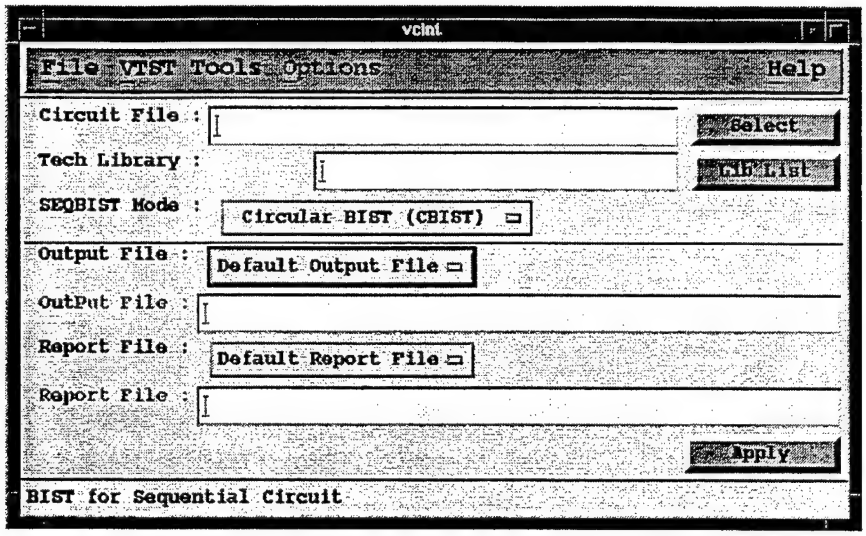


Figure 14.23: VTST sequential BIST

Partial Scan, *Full Scan*, and *Partial Scan with CBIST*. *Circular BIST* is the BIST design using circular self-test path for sequential circuit. *Partial Scan* is scan design with selected D-flipflops in scan chain. *Full Scan* is scan design with all D-flipflops in scan chain. *Partial Scan with CBIST* is a combination of *CBIST* and *Partial Scan*.

- **Default Input Tech Library (.inlib) and Tech Path (.libpath) :** If users do not provide any information about the input tech library (leave those field blank), SEQBIST will look up *.inlib* file to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in current working directory (refer to the Section 14.3.2 for how to set up *.inlib* and *.libpath* file).

14.4.7 Autonomous

VTST AUTONOMOUS tool is used to partition a circuit into two or more sub-circuits. To invoke the AUTONOMOUS window, click

VTST Tools ⟶ Basic ⟶ AUTONOMOUS

the *vclnt* window will change to AUTONOMOUS mode as shown in Figure 14.24.

The following is the explanation for each field on this window :

- **Circuit File :** This field tells AUTONOMOUS what is the circuit that AUTONOMOUS will work on.

- **Input Tech Lib :** This field tells the AUTONOMOUS which tech library of the input circuit is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).

- **Mode :** This option menu allows users to choose different methodologies to partition a circuit. The available methodologies include *Level contour partitioning*, *Dependency contour partitioning*, and *LFSR Size partitioning*. *Level contour partitioning*

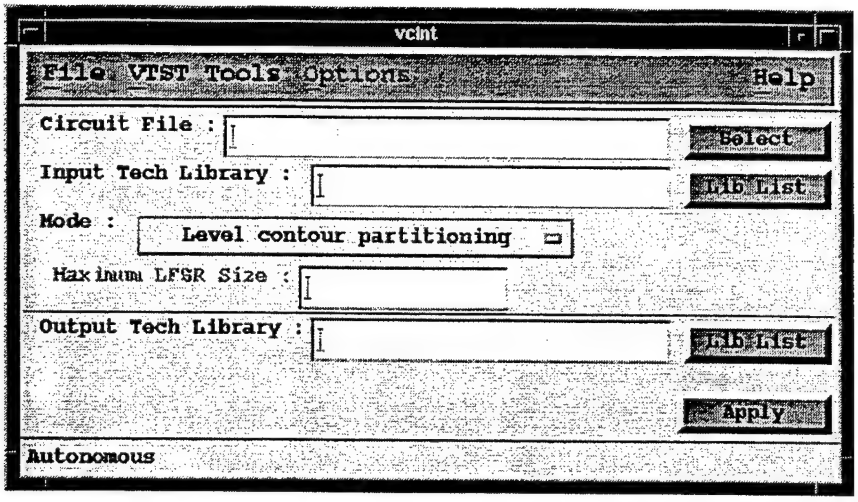


Figure 14.24: VTST Autonomous

will partition a circuit into two subcircuits according to the specified distance from the primary inputs. *Dependency contour partitioning* will partition a circuit into two subcircuits according to the specified primary output dependency. *LFSR Size partitioning* will partition a circuits into several partitions according to dependency specified by users. This option will guarantee that all the partitions will have maximum dependency less than or equal to the specified maximum LFSR size.

- **Maximum LFSR Size :** If users choose *LFSR Size partitioning* methodology, this field is used to specify the maximum size of LFSR that is allowed to apply on all sub-circuits.

- **Output Tech Lib :** This field tells the AUTONOMOUS which tech library that the sub-circuits are used. Users can type in the valid library name in the text input field, press [Lib List] button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default output library (*.outlib*)).

- **Default Tech Library (.inlib and .outlib) and Tech Path (.libpath) :** If users do not provide any information about the input/output tech library (leave those field blank), the parser will look up *.inlib* and *.outlib* files to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in current working directory (refer to the Section 14.3.2 for how to set up *.inlib*, *.outlib*, and *.libpath* files).

14.4.8 BISTSYN

VTST BISTSYN tool is a test signal reduction synthesizer for pseudo-exhaustive testing. To invoke the BISTSYN window, click

[VTST Tools] ⟶ [Basic] ⟶ [BISTSYN]

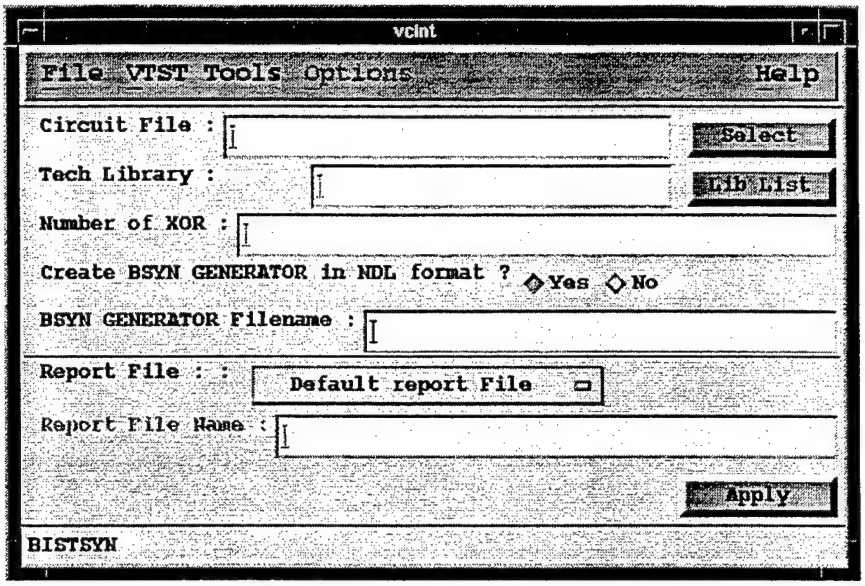


Figure 14.25: VTST BISTSYN

the *vclnt* window will change to BISTSYN mode as shown in Figure 14.25.

The following is the explanation for each field on this window :

- **Circuit File :** This field tells BISTSYN what is the circuit that BISTSYN will work on.
- **Input Tech Lib :** This field tells the BISTSYN which tech library of the input circuit is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).
- **Number of XOR :** Users can specify the maximum XOR level in this field. BISTSYN will generate XOR trees with the level less or equal than the maximum XOR level.
- **Create BISTSYN GENERATOR in NDL format :** A BISTSYN generator in NDL netlist format will be generated if users choose *Yes* in this option menu. If users choose BISTSYN as a synthesis path and wants to use BUILDER to create final BISTed design, then the BISTSYN generator in NDL netlist format needs to be generated.
- **BISTSYN GENERATOR Filename :** If users want BISTSYN to generate a BISTSYN generator in NDL netlist format, then users can specify the file name in this field.
- **Default Input Tech Library (.inlib) and Tech Path (.libpath) :** If users do not provide any information about the input tech library (leave those field blank), SEQBIST will look up *.inlib* file to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in current working directory (refer to the Section 14.3.2 for how to set up *.inlib* and *.libpath* file).

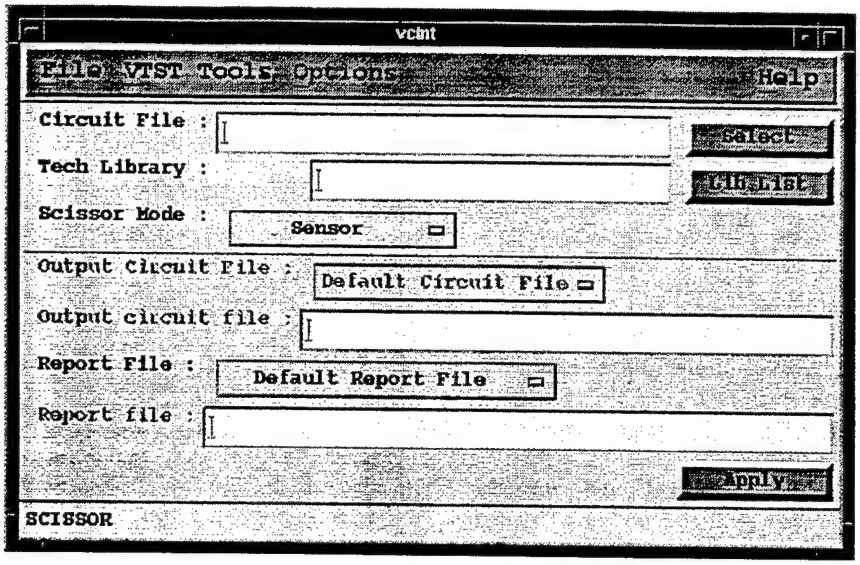


Figure 14.26: VTST Scissor

14.4.9 Scissor

VTST SCISSOR tool provides several different methodologies to preprocess a circuit. To invoke SCISSOR window, click

VTST Tools ⟶ Basic ⟶ SCISSOR

the *vclnt* window will change to SCISSOR mode as shown in Figure 14.26.

The following is the explanation for each field on this window :

- **Circuit File :** This field is used to specify the circuit that SCISSOR will work on.
- **Input Tech Lib :** This field tells the SCISSOR which tech library that the input circuit is used. Users can type in the valid library name in the text input field, press Lib List button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).
- **Scissor Mode :** This option menu allows users to choose different methodologies to preprocess a circuit. The options are *Sensor*, *Regular*, *Register free*, *DFF Free*, and *Feedback Free*. *Sensor* will report the nature of the test circuit. *Regular* will remove all the untestable components such as tri-state buffer, unknown components, and bi-direction I/O and synchronize all control signals such as clock, clear, and set for DFFs and latches. *Register free* will remove all the register components (DFF and latch) from the circuit. *DFF Free* will remove all DFF from the circuit. *Feedback Free* will remove all the register components that are in feedback path from the circuit.
- **Default Input Tech Library (.inlib) and Tech Path (.libpath) :** If users do not provide any information about the input tech library (leave those field blank), SEQBIST will look up *.inlib* file to get the default library. Users must provide the tech lib path in *.libpath* file, if the tech lib is not in the current working directory (refer to the Section 14.3.2 for how to set up *.inlib* and *.libpath* file).

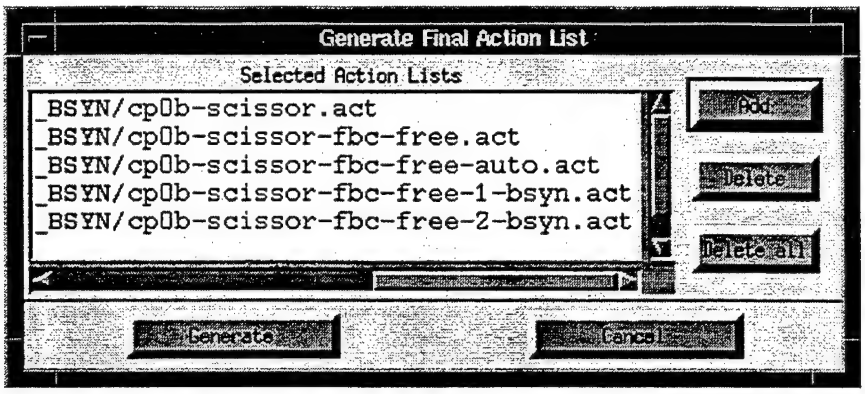


Figure 14.27: VTST action generator

- **Unknown Component List (.ucomplst) :** VTST SCISSOR needs to know which components are unknown so SCISSOR can remove those components from the test circuit. This information is provided in *.ucomplst* file (refer to the Section 14.3.5 for how to set up *.ucomplst* file).

14.4.10 Action Generator

Action Generator (*ActGEN*) is the action consolidated utility for VTST. To invoke *ActGEN* window, click

VTST Tools ⟶ Basic ⟶ VTST BUILDER ⟶ Create Action List

or

Files ⟶ Action Generator

the *Generate Final Action List* dialog window will pop up as shown in Figure 14.27.

This dialog window contains a file list window, users must place the action lists files according to the sequence of applied synthesis tools in this window. Please see Section 14.3.3 for how to modify this file list.

After all the action list files are placed in the file list window sequentially, users can hit the Generate button to invoke *ActGEN* utility to create the final consolidated action list file.

14.4.11 Xor Tree Generator

Xor Tree Generator is a VTST utility which can generate a 64 bits output XOR tree in TDN format. The purpose for this XOR Tree is to reduce the test circuit's output size during testing. To invoke *Xor Tree Generator* window, click

VTST Tools ⟶ Basic ⟶ Fault Simulator ⟶ Generate XOR Array

or

Files ⟶ XOR Generator

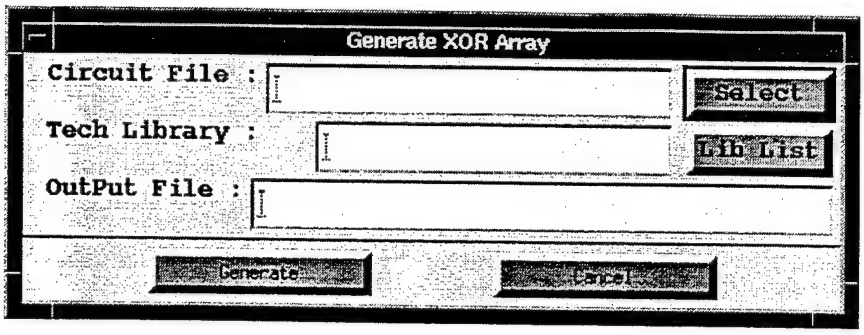


Figure 14.28: VTST XOR tree generator

the *Generate XOR Array* dialog window will pop up as shown in Figure 14.28.

The following is the explanation for each field on this window :

- **Circuit File :** This field is used to specify the circuit which need a XOR Tree on the output side.
- **Input Tech Lib :** This field tells the generator which tech library that the input circuit is used. Users can type in the valid library name in the text input field, press [Lib List] button and select a library from the *lib list* dialog window (or leave this field blank to let parser to use the default input library (*.inlib*)).
- **Output File :** The newly generated XOR Tree in TDN format will be stored in the file specified in this field.

After fills in the information, users can press the [Generate] button to invoke the tool to generate the XOR Tree.

14.4.12 CMDE Simulator Control files Generator

CMDE Simulator Control files Generator *CmdeGEN* is a VTST utility used to generate the CMDE simulator control and pattern files for testing the BISTed circuit on test mode. To invoke *CmdeGEN* window, click

[Files] ⟶ [CMDE Generator]

the *CMDE generate* dialog window will pop up as shown in Figure 14.29.

The following is the explanation for each field on this window :

- **FSIM Information files :** VTST FSIM tool will provide a simulation information file, which is used by *CmdeGEN* to generate the CMDE simulator control and pattern files. FSIM will be applied to all sub-circuits and generate more then one simulation information files. Users must select all the simulation information files and place them in the file list window according to the action's sequence.
- **Is CBIST Involved :** If CBIST is involved for the final BIST design, a specified control signal is needed. If this option menu is set to *Yes*, *CmdeGEN* will create this specified control signal in both control and pattern files.

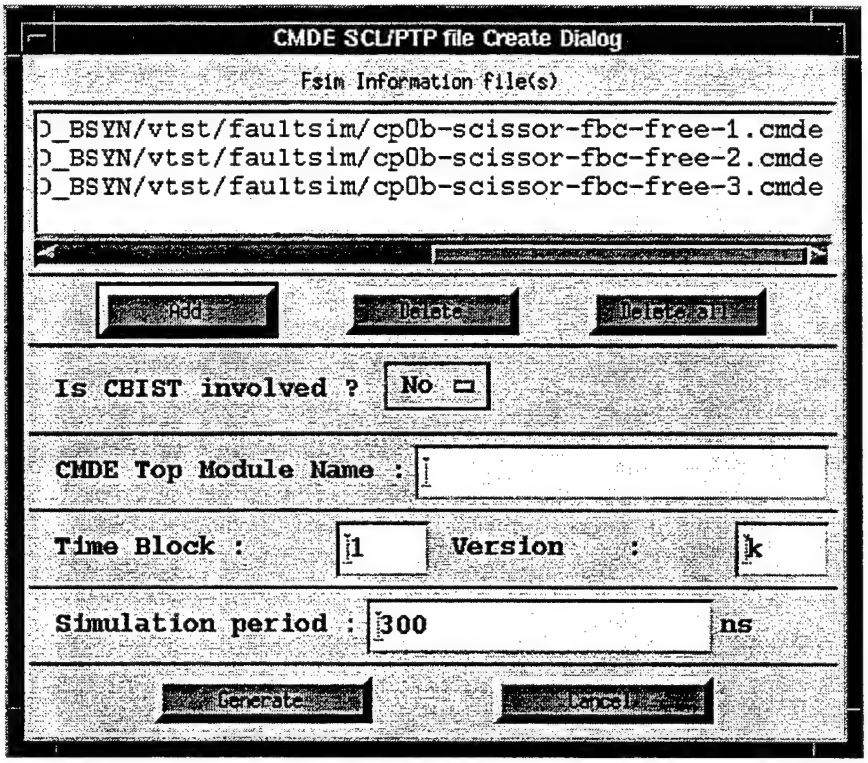


Figure 14.29: VTST CMDE simulation files generator

- **CMDE Top Module Name :** This filed specifies the top module name, this name will be used as a part of the filename for both control and pattern files. Users need to provide the same top module name when CMDE is invoked.
- **Time Block and Version :** These two fields will be used as parts of the filename for both control and pattern files. Users need to provide the same *Time Block* and *Version* when CMDE simulator is invoked.
- **Simulation Period :** This field is used to specify the time period for a clock cycle. The unit is nanosecond.

After filling in all the information, users can press the Generate button to invoke the tool to generate the cmde simulator control and pattern files.

Chapter 15

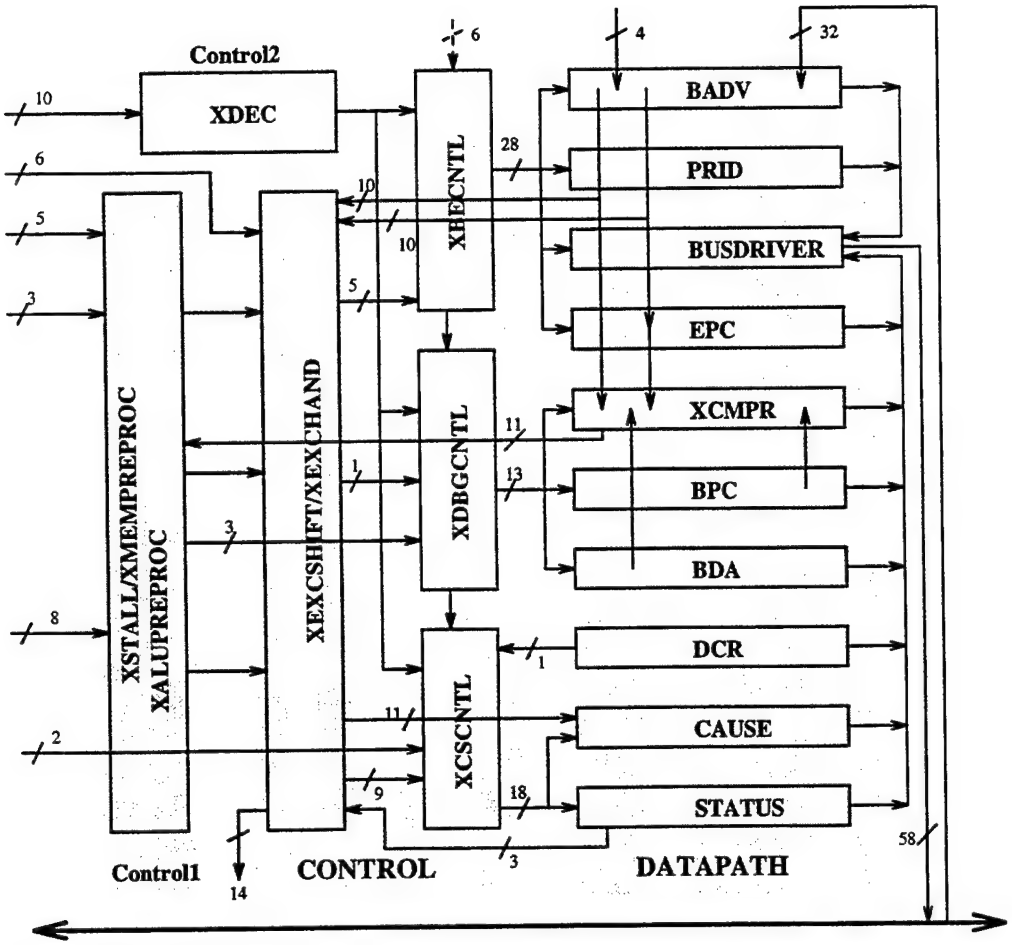
Test Case: System Coprocessor CP0 of the LR33000 CPU Core

15.1 An Overview of CP0 of the LR33000 CPU Core

The system control coprocessor (CP0) of the LR33000 CPU core is incorporated in the MIPS R3000 chip to support the virtual memory system and exception handling functions of the CPU. The CP0 is responsible for the low level exception handling: prioritizing exceptions, saving exception information for the exception handler, flushing the CPU pipeline, and transferring control to exception handler. The total gate count of CP0 is 5899. The number of primary inputs and primary outputs of CP0 are 138 and 85 respectively. Figure 15.1 shows a block diagram of CP0 which depicts the data path and the control of CP0. The CP0 datapath consists of 10 blocks of registers: Bad Virtual Address Register (BADVR), Breakpoint Data Address Register (BDAR), Breakpoint Program Counter Register (BPCR), Data Bus Transceiver (BUSDRIVER), Cause Register (CAUSER), EPC Register (EPCR), Processor ID Register (PRIDR), Address comparators (XCMPR), Stutus Register (XSTATR), Value Added Registers (XVAR). These registers support all the exception handling functions of the CPU core. The control section of CP0 consists of 9 modules: preprocessing for ALU exception signals (XALUPREPROC), BADVR and EPCR control (XBECNTL), CAUSER and STATR control (XCSCNTL), BDAR and BPCR control (XDBGCNTL), CP0 instruction decode (XDEC), exception handling logic (XEXCHAND), exception invalidation logic (XEXCSHIFT), preprocessing for MEM exception signals (XMEMPREPROC), and RUN/STALL control logic (XSTALL).

15.2 VTST Parser on CP0

The CP0 was originally designed in the C-MDE design environment by the LSI Logic Corp. and is written in NDL. The hierarchical NDL netlist of CP0 was first read in by the VTST NDL Parser following the NDL Parser system flow. And, the VTST NDL Parser flattened the design and generated the TDN format for BIST design and testability analysis.



DATHPATH includes :

BADVR -- Bad Virtual Address Register

BDAR -- Breakpoint Data Address Register

BPCR -- Breakpoint Program Counter Register

BUSDRIVER -- Data Bus Transceiver

CAUSER -- Cause Register

EPCR -- Exception Program Counter Register

PRIDR -- Processor ID Register

XCMPR -- Address comparators

XSTATR -- Status Register

XVAR -- Value Added Register

Control1 includes:

XSTALL -- RUN/STALL control logic

XMEMPREPROC -- preprocessing for MEM exception signals

XALUPREPROC -- preprocessing for ALU exception signals

XEXCSHIFT -- exception invalidation logic

XEXCHAND -- exception handling logic

Control2 includes:

XDEC -- CP0 instruction decoder

XBECNTL -- BADVR and EPCR control

XDBGCNTL -- BDAR and BPCR control

XCSCNTL -- CAUSER and STATR control

Figure 15.1: Block Diagram of the system control of coprocessor (CP0) of the LR33000 CPU

We use the distributed VTST system and the window environment to create four files; They are *.file*, *.inlib*, *.outlib*, and *.ucomplist.* The *.file* file which includes all the files that are related to the CP0 design and all the equivalent macro cells which have been formatted. The *.inlib* and *.outlib* files indicate the LSI Logic LCB007 cell library as the input and output technology library to be used. There are three macro cells in the design that do not have equivalent logic. These three elements are the pre-charge elements *prechb* and *prechc* and the bus holder *bhd1a.* Therefore, *prechb*, *prechc*, and *bhd1a* are listed in *.ucomplist* file.

15.3 Circuit Analysis/Preprocessing Tool on CP0

First of all, we employ the circuit analyzer/preprocessor - Sensor/Scissor on CP0, The circuit analyzer - Sensor reports the following. There are

- 138 primary inputs
- 85 primary outputs
- 90 latches
- 420 D-type flip-flops
- 1860 all other gates including tristate buffers and inverters
- 32 bidirectional inputs/outputs
- 121 D-type flip-flops are in feedback paths
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 64 tristate buffers having the data inputs from the same source and the enable signals from different sources
- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 288 tristate buffers having the data inputs from different sources and the enable signals from different sources.

After that, circuit preprocessor restructures the CP0 and generates a testable CP0 in the TDN format called *cp0b-scissor.tdn.* The characteristics of the circuit after Scissor are:

- 140 primary inputs
- 6 control inputs
- 287 primary outputs
- 90 latches

- 420 D-type flip-flops
- 703 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources
- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 139 Most Significant Bits (MSB) and 1 in the Least Significant Bits(LSB), and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 80.96%.

15.4 Pseudo-Scan Scissor on CP0

15.4.1 Pseudo-Scan with no Flip-Flops and Latches

Pseudo-Scan Scissor with RegFree is applied to *cp0b-scissor.tdn*. A testable circuit with no flip-flops and latches is generated. The name of the file is *cp0b-scissor-regfree.tdn*. The characteristics of the circuit after Pseudo-Scan Scissor are:

- 383 primary inputs
- 6 control inputs
- 286 primary outputs
- 0 latches
- 0 D-type flip-flops
- 697 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources

- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 382 MSB and 1 in the LSB, and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 57.00%.

15.4.2 Pseudo-Scan with no Flip-Flops and without Latches in Feedback path

Pseudo-Scan Scissor with flipflop-Free is applied to *cp0b-scissor.tdn*. A testable circuit with no flip-flops and without latches in the feedback path is generated. The name of the file is *cp0b-scissor-ff-free.tdn*. The characteristics of the circuit after Pseudo-Scan Scissor are:

- 332 primary inputs
- 6 control inputs
- 280 primary outputs
- 0 latches
- 37 D-type flip-flops which are those D-type flip-flops with power or ground as inputs
- 703 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources
- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 331 MSBs and 1 in the LSB, and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 55.21%.

15.4.3 Pseudo-Scan with no Flip-Flops and no Latches in Feedback Path

Pseudo-Scan Scissor with RegFree is applied to *cp0b-scissor.tdn*. A testable circuit with no flip-flops and latches in the feedback path is generated. The name of the file is *cp0b-scissor-fbfree.tdn*. The characteristics of the circuit after Pseudo-Scan Scissor are:

- 156 primary inputs
- 6 control inputs
- 298 primary outputs
- 90 latches
- 404 D-type flip-flops
- 703 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources
- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 155 MSBs and 1 in the LSB, and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 87.00%.

15.5 BISTSYN on CP0 After Pseudo-Scan Scissor on CP0

15.5.1 Pseudo-Scan with no Flip-Flops and Latches

Pseudo-Scan Scissor with RegFree is applied to *cp0b-scissor.tdn*. A testable circuit with no flip-flops and latches is generated. The name of the file is *cp0b-scissor-regfree.tdn*. The characteristics of the circuit after Pseudo-Scan Scissor are:

- 383 primary inputs
- 6 control inputs

- 286 primary outputs
- 0 latches
- 0 D-type flip-flops
- 697 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources
- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 382 MSBs and 1 in the LSB, and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 57.00%.

15.5.2 Pseudo-Scan with no Flip-Flops and without Latches in Feedback Path

Pseudo-Scan Scissor with Flipflop-Free is applied to *cp0b-scissor.tdn*. A testable circuit with no flip-flops and no latches in the feedback path is generated. The name of the file is *cp0b-scissor-ff-free.tdn*. The characteristics of the circuit after Pseudo-Scan Scissor are:

- 332 primary inputs
- 6 control inputs
- 280 primary outputs
- 0 latches
- 37 D-type flip-flops which are those D-type flip-flops with power or ground as inputs
- 703 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources

- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 331 MSBs and 1 in the LSB, and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 55.21%.

15.5.3 Pseudo-Scan with no Flip-Flops and no Latches in Feedback Path

Pseudo-Scan Scissor with Feedback-Free is applied to *cp0b-scissor.tdn*. A testable circuit with no flip-flops and latches in the feedback path is generated. The name of the file is *cp0b-scissor-fbfree.tdn*. The characteristics of the circuit after Pseudo-Scan Scissor are:

- 156 primary inputs
- 6 control inputs
- 298 primary outputs
- 90 latches
- 404 D-type flip-flops
- 703 all other gates excluding tristate buffers and inverters
- 355 inverters
- 0 tristate buffers having the data inputs from the same source and the enable signals from the same source
- 0 tristate buffers having the data inputs from the same source and the enable signals from different sources
- 0 tristate buffers having the data inputs from different sources and the enable signals from the same source
- 0 tristate buffers having the data inputs from different sources and the enable signals from different sources.

With the initial seed of all 0's in the 155 MSBs and 1 in the LSB, and 1024 test patterns generated by the LFSR, the fault coverage after fault simulation is 87.00%.

15.6 Experimental Results

```
*******************************
CIRCUIT NATURE OF CP0 BY SENSOR
*******************************
Circuit                 : cp0b
Primary Inputs          : 138
Control Inputs          : 0
Primary Outputs         : 85
Latches                 : 90
Flip flops              : 420
Instances               : 1858
Bidirection IO          : 32
Tristate buffers having the Same Data and Control signals are : 0
Tristate buffers having the Same Data and Different Control signals are : 64
Tristate buffers having Different Data and the Same Control signals are : 0
Tristate buffers having Different Data and Different Control signals are : 288

***********************************
TESTABLE STRUCTURE OF CP0 BY SCISSOR
***********************************
Design: cp0b-scissor.tdl
PI: 138
CPI: 2
PO: 289
latch: 90
flip-flop: 420
inverter: 357
tristates: 0
other: 699
tool time: 5.533 sec
total time: 6.150 sec

Denote: Seed = Initial seed in hexadecimal
        Pttn = Total # of test patterns
        F = Total # of faults in circuit
        U = # undetected faults in circuit
        FC = Fault Coverage = # Faults Detected / Total # Faults
        Time/s = run time in second in Sparc 10 model 30 workstation
```

Seed(hex)	Pttn	F	U	FC	Time/s
10af56dd0c082fcd11011	262144	3898	413	89.40%	4702.033
26ad	16384	3898	465	88.07%	300.283
10af56dd0c082fcd11011	16384	3898	469	87.97%	303.600

```
*****************************
FAULT GRADING RESULTS OF CBIST
*****************************
Design: cp0b-scissor-cbist.tdl
PI: 138
CPI: 2
PO: 289
flip-flop: 420
latch: 90
inverters: 357
tristates: 0
other gates: 862
tool time: 6.433 secs
total time: 6.683 secs
```

Seed(hex)	Pttn	F	U	FC	Time/s
4b1	262144	4974	317	93.35%	4786.683
4b1	16384	4764	345	92.76%	303.450
1	262144	4974	365	92.66%	4702.033
8c01	16384	4974	369	92.58%	264.283
10f234afb0100bdc0178234cffffdf0045	16384	4974	377	92.42%	268.733
10101010101010101010101010101010ff ff34567823	16384	4974	387	92.22%	268.050
8c01	16384	4856	345	92.90%	278.333
10f234afb0100bdc0178234cffffdf0045	16384	4856	349	92.81%	268.317
10101010101010101010101010101010ff ff34567823	16384	4856	349	92.81%	268.750
10101010101010101010101010101010ff ff34567823	32768	4856	342	92.96%	525.980
10ffffff82fcd1102f2c011	16384	4856	358	92.63%	271.517
10ffffff82fcd1102f2c011	16384	6122	546	91.08%	291.717
10ffffff82fcd1102f2c011	32768	6122	524	91.44%	569.650
4b1	131072	6122	522	91.47%	2287.250
4b1	262144	6122	473	92.27%	4614.167

```
*************************************
FAULT GRADING RESULTS OF REGISTER_FREE
*************************************

(A) W/O BISTSYN
```

Seed(hex)	Pttn	F	U	FC	Time/s
804186104	16384	2865	557	80.56%	105.017
804185831	16384	2865	569	80.14%	

804185643	16384	2865	572	80.03%
803757281	16384	2865	573	80.00%

(B) W. BISTSYN (run time 91.600 sec)

Seed(hex)	Pttn	F	U	FC	Time/s
8000100040000a	16384	2815	354	87.42%	203.450
3000300100	16384	2815	356	87.35%	204.750
000500000700009000	16384	2815	357	87.32%	204.200
80001	16384	2815	358	87.28%	203.950
1000001000000200000400001	16384	2815	359	87.25%	207.483

****
FAULT GRADING RESULTS OF FLIP-FLOP_FREE
****

(A) W/O BISTSYN

Seed(hex)	Pttn	F	U	FC	Time/s
803765370	16384	3326	783	76.46%	138.317

(B) W. BISTSYN (run time 6.667 sec)

Seed(hex)	Pttn	F	U	FC	Time/s
10101010000000000000000001000000	16384	3224	477	85.20%	235.667
20012dfc00000000000000000000345	16384	3224	487	84.89%	236.600
800000008000000000800000ffffffff 9090222	16384	3224	492	84.74%	241.250

****
FAULT GRADING RESULTS OF FEEDBACK_FREE
****

(A) W/O BISTSYN

Seed(hex)	Pttn	F	U	FC	Time/s
1578864788855555889fffcccdccdcffd	262144	3978	358	91.00%	4726.200

d5666

44a3	16384	4064	405	90.03%	268.950
1201	16384	4064	445	89.05%	293.550

(B) W. BISTSYN (run time 11.900 sec)

Seed(hex)	Pttn	F	U	FC	Time/s
111100000ffffff	16384	3872	376	90.29%	298.233
3	262144	3872	366	90.55%	4434.200
3	16384	3872	382	90.13%	290.200
3	16384	3876	381	90.17%	257.683
8c	16384	3876	391	89.91%	291.883

****
PARTITIONING RESULTS BY FEEDBACK_FREE, AUTONOMOUS AND BISTSYN
****
Circuit : CP0
Design : cp0b-scissor-fbc-free
Number of subcircuits: 3
Number of edge cut: 125
Memory used : 32725537
Max Dependency : 18

****
Sub-circuit: cp0b-scissor-fbc-free-1
****
PI : 144
CPI : 11
PO : 285
ngate: 1076
Max Dependency : 18
Time to run BISTSYN : 9.550 secs

(A) W/O BISTSYN

Seed(hex)	Pttn	F	U	FC	Time/s
1afa1	16384	2836	287	89.88%	229.55
10101456fff	16384	2840	306	89.23%	237.98
1	16384	2840	347	87.78%	258.13

(B) W. BISTSYN

```
-------------------------------------------------------------------------------
Seed(hex)                       Pttn     F       U       FC        Time/s
-------------------------------------------------------------------------------
1afa1                          16384    2766     331     88.03%    229.55
10101456fff                    16384    2770     297     89.28%    215.18
fff0f                          16384    2770     298     89.24%    214.07
-------------------------------------------------------------------------------

************************************
Sub-circuit: cp0b-scissor-fbc-free-2
************************************
PI : 78
CPI : 2
PO : 37
ngate: 149
Max Dependency : 10
Time to run BISTSYN :                  1.700 secs

(A) W/O BISTSYN

-------------------------------------------------------------------------------
Seed(hex)                       Pttn     F       U       FC        Time/s
-------------------------------------------------------------------------------
1                              16384     355     2       99.44%    26.017
-------------------------------------------------------------------------------

(B) W. BISTSYN

-------------------------------------------------------------------------------
Seed(hex)                       Pttn     F       U       FC        Time/s
-------------------------------------------------------------------------------
1                               1024     357     2       99.44%     1.350
-------------------------------------------------------------------------------

************************************
Sub-circuit: cp0b-scissor-fbc-free-3
************************************
PI : 134
CPI : 6
PO : 57
Flip-flops :                       39
Latches : 10
Inverter :  71
Tristates :  0
Other instances : 209
Max Dependency : 18
Time to run BISTSYN :                  4.633 secs

(A) W/O BISTSYN
```

Seed(hex)	Pttn	F	U	FC	Time/s
4b1	16384	786	23	97.07%	81.667

(B) W. BISTSYN

Seed(hex)	Pttn	F	U	FC	Time/s
4b100	16384	778	87	88.82%	78.550
107ddff07060	16384	778	86	88.95%	76.617

****
* BISTed CP0 RESULTS *
****

cp0b original : 18008 c.u. (cell Unit)

Library: LCB007

Method	with mux only (c.u./overhead)	with TPG/MISR /XorTree/mux (c.u./overhead)	BUILDER CPU time time (sec)	pttn	FC
SCISSOR	20429 (11.85%)	34552 (47.88%)	14.38	262144	89.40%
CBIST	22548 (20.13%)	36671 (50.89%)	15.83	262144	93.35%
FBC_free	20906 (13.86%)	35892 (49.83%)	15.80	262144	91.00%
FBC_free_bsyn	20906 (13.86%)	31179 (42.24%)	15.88	262144	90.55%
FF_free_bsyn	23165 (22.26%)	33709 (46.58%)	19.65	16384	85.20%
Autonomous* (FBC_free_bsyn)	22031 (18.26%)	31805 (43.38%)	20.10	33792	90.14%

****
* Example: BUILDER results after applying Autonomous and FBC_free_bsyn.
**********8**********

GATE COUNT BEFORE MODIFY : 18008

```
GATE COUNT ATFER MODIFY : 22031
OVERHEAD is : 0.182606

BUILDER now is creating 18 TPG_array
GATE COUNT for TPG : 985

BUILDER now is creating 64 MISR_array
GATE COUNT for MISR : 4429
GATE COUNT for MUX before MISR : 1040
GATE COUNT for  a XOR  array: 1989
GATE COUNT for all XOR arrays :  1989
GATE COUNT for test control logics: 1331

Create New NDL Net List file
GATE COUNT TOTAL : 31805
OVERHEAD is : 0.433800

******* VTST BUILDER IS COMPLETE *******
User Time ......... 18.333333 seconds
System Time ....... 1.766667 seconds
Total Time ........ 20.100000 seconds

Note: The MUX is counted 15 c.u.  (equivalent to three logic gates - worst case).
```

Chapter 16

TEST CASE: ENHANCED MEMORY CHIP (EMC)

16.1 Introduction

This project analyzes the structure of the Enhanced Memory Chip (EMC) to a level sufficient for the testing of each module. It also includes the testing of each module at module level and finally the testing of three main components (Multiplier Tree, ALU Block and Memory Plane) of the chip at the chip level.

In every computer graphics system, there is some interaction between a special memory called a frame buffer and a computer engine. It is the architecture between these two that determines how fast, flexible, and expensive the graphics subsystem is. An approach used by the reseachers at the University of North Carolina is shown in Figure 16.1.

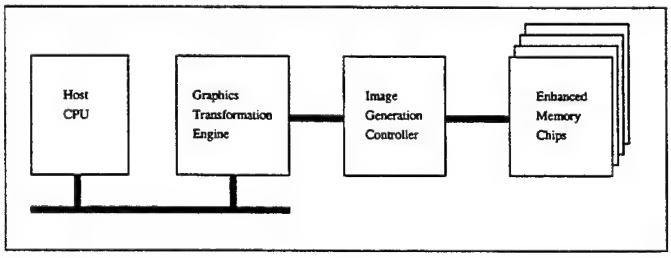


Figure 16.1: A typical Computer Graphics System Architecture

In this architecture, the host CPU runs the user's application which edits a display list. This display list contains graphics primitives to be performed on the display. The graphics processor performs preliminary transformations on the primitives. Following this, the transformed data and operations are sent to the image generation controller(IGC). The IGC is a microcode sequencing engine which controls the enhanced memory chips. Given simple graphics instructions from the graphics processor, it will pipeline and control

the data and instructions fed to the enhanced memory chips. The enhanced memory chip (EMC) is fundamentally memory that is enhanced with tightly-coupled computational logic on the same chip. The logic is arranged in a single instruction, multiple data architecture that can efficiently perform the linear expression evaluation that is common in the lowest level graphic rasterization. Testing of the EMC is of interest in this project.

The objective of this project is to develop a set of test vectors for the three main components of the EMC. Two sets of test vectors will be produced: one with the minimum set of vectors and a second set offering the highest possible fault coverage for the EMC using the scan chain scenario.

16.2 EMC top level description

An overview of the basic architecture of the EMC is shown in Figure 16.2. Detailed descriptions of each of the major subsystems will follow in the subsequent sections. Pipelined, serialized coefficients (A, B and C Coef) are fed into a linear expression evaluator tree (Multiplier Tree). The 64 bit outputs (TreeData) of the tree are the serialized values of Ax+By+C. A, B, and C Coef are the serialized coefficients fed from the IGC, and x and y are the unique locations of each pixel memory on the chip. The outputs (TreeData) of the tree are used by a bank of pixel arithmetic-logic units (ALU's).

The ALU's can manipulate the values read from the tree via TreeData and memory via ALURdData, can compute new values, and can store the results into the pixel memory via ALUWrData lines. There is one ALU per pixel memory location. Because of this, each ALU is very simple and small. While the same instructions (AControl, BControl and CControl) are fed to all ALU's, memory write operations can be made conditional through the ForceEnable line so that only desired pixels get updated.

The memory has effectively two ports. Each port is made up of 64 cells and each cell contains up to 64 bits per word. One port can be read or written by the ALU's while other can be accessed by the input/output data pins of the chip. The selection of the ports and cells for ALU or input/output are done by the rowselect line, wordlines and IORdEn lines to the Memory Plane respectively. The Word Decoder is used to decode the 6 Wordlines into 64 Wordlines to the Memory. The Memory Out Controller is actually a 64 bit counter that will increment at each clock to select each cell of the input/output port. The IO_OutEn will enable the content of the input/output to be released via 32 IORdData lines to the outside. This is how the frame buffer can be preloaded and also how the results are read out by the video rasterization subsystem.

Within the Multiplier Tree and the ALU block, there are 128 scan registers connected in a scan chain. This scan chain enables Multiplier Tree, ALU Block and Memory Plane to be tested independently.

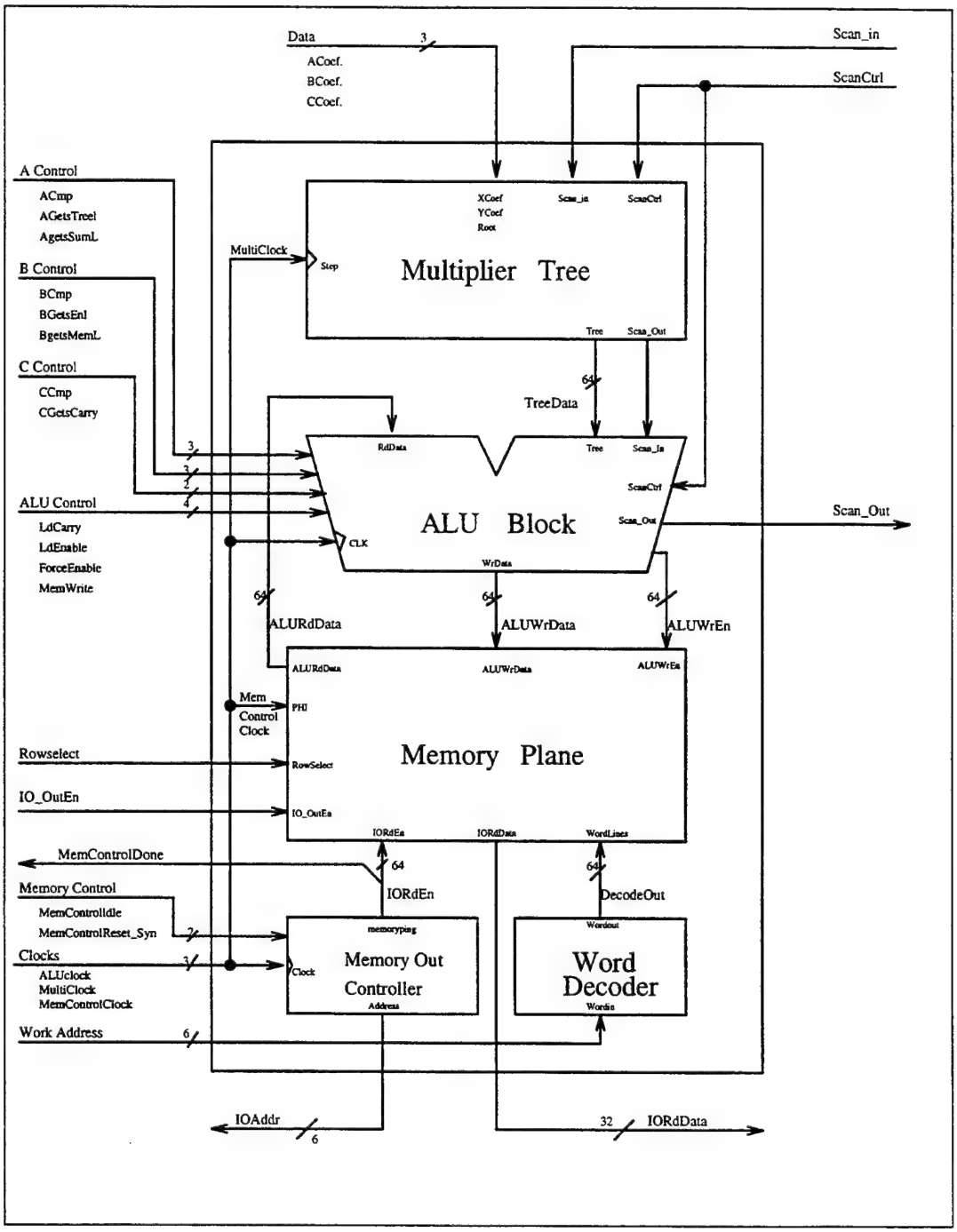


Figure 16.2: Basic Architecture of Enhanced Memory Chip

16.3 Multiplier Tree Description and Module Testing

The Multiplier Tree is essentially a parallel, bit-serial multiplier, taking in data from three bit-serialized streams of A, B, and C coefficients (chip label) or XCoef, YCoef and Root respectively (module label). Its outputs are the function F(x,y) = Ax+By+C, where x and y are the coordinates of the pixel on the display. These outputs (64 bits) are fed into the pixel ALU's via the ALU's Tree inputs.

16.3.1 Multiplier Tree Description

The layout of a single multiplier stage is shown in Figure 16.3.

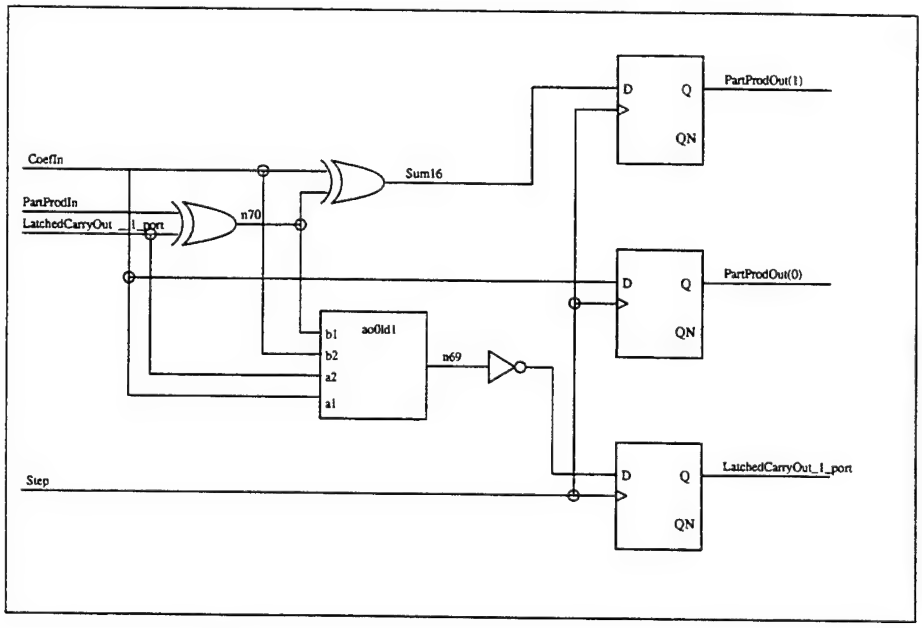


Figure 16.3: Single Multiplier Stage

The key element of a Multiplier Tree is a full adder. The Ain input of the adder gets the serialized coefficients of A or B from the expression Ax+By+C. The Bin input of the adder gets the appropriate result (PartProdIn) from the previous multiplier stage in the tree. The Cin input of the adder gets the result of the previous stage's Cout (LatchedCarryOut_1_port). Three flip-flops are selected to store the results for the next stage of the tree. The first flip-flop stores the adder's Sum output. The second flip-flop bypasses the adder and stores the partial product from the previous stage directly. The third flip-flop stores the carry out bit generated for the next stage computation.

The layout of a depth 2 multiplier is shown in Figure 16.4. The components used here are twice of that used in the single multiplier stage. A close look at it reveals that it is made

up of 2 1-bit full adders producing two sum results (PartProdOut(2) and PartProdOut(3)) and two carry out results (LatchedCarryOut_2_port and LatchedCarryOut_3_port). These results are stored in four flip-flops. Two other flip-flops store the value of PartProdOut(0) and PartProdOut(1) from the previous single multiplier stage. The description of a depth 4, 8 and 16 multipliers are similar to the depth 2 multiplier described above, except that the latter multipliers have components twice as many as their previous stage multiplier. The outputs produced by the latter multipliers are also twice as many as their previous stage multiplier.

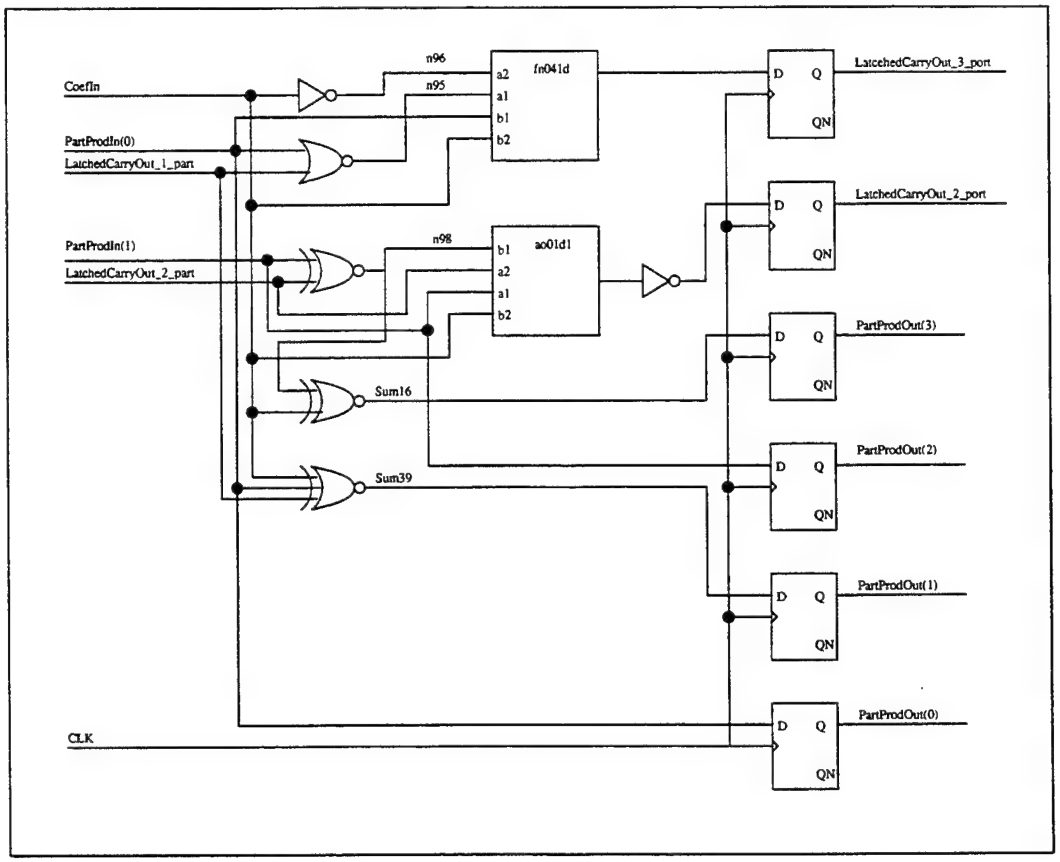


Figure 16.4: Depth 2 Multiplier Stage

The last stage of the multiplier which is depth 32 multiplier is slightly different from those discussed above The details of this multiplier are shown in Figure 16.5 and Figure 16.6. Figure 16.6 shows that the multiplier contains 32 full adders which produce 32 bits of sum results (32 bits TreeData) and 32 carry bits (LatchedCarryOut_1_port to LatchedCarryOut_32_port). The other 32 bits TreeData are results passed from the previous depth 16 multiplier. Notice the presence of 32 multiplexers and 'OR' gates before all the flip-flops, actually form a scan chain. A high at the ScanCtrl pin will disable the scan process and allow the Tree results to be output to the ALU Block in parallel. When the ScanCtrl pin is low with the presence of clock signal (Step), the flip-flops will act as shift registers shifting the data of each flip-flop serially out to the Scan_Out line to the ALU Block's Scan_In line. This scan chain serves two purposes; it can be used to scan out the results of the Multi-

plier Tree or to scan in test data to the ALU Block to test the ALU and the Memory Plane.

The layout of the whole Multiplier Tree is shown in Figure 16.7. It is made up of six different stages namely MutliStage Depth 1, 2,4, 8, 16 and 32. Its inputs are XCoef, YCoef and Root. Its outputs are 64 bits Tree (0:63) from the MultiStage Depth 32.

16.3.2 Multiplier Tree Module Testing

A VHDL program named test_multtree_struct_scan.vhd is used to test the Multiplier Tree at the module level. There are only three primary inputs present in this module. They are XCoef, YCoef and Root. Sixty-five sets of test data are provided by multtree_test.vectors to assist in the testing of the module. These sixty-five sets of test data are generated by the VLSI Testability Synthesis Tool (VTST) Test generation program.

Below is a brief description of the test_multtree_struct_scan.vhd program. The program starts off with presetting the values of YCoef, XCoef, Root and then clock the MutliClock (Step) thirteen times to initialize all the flip-flops in all the MultiStages to a predefined value. The reason for this is that the outputs from the MultiStage Depth 1 stage takes about ten clocks to ripple to MultiStage Depth 32 stage. Thus the initial thirteen clock cycles of results produced by the Multiplier Tree are random. Defined valid results appear only after the thirteen clock cycles. After this, the program will read the first set of data in the sequence of Root, XCoef and YCoef from the multtree_test.vectors files. These data are assigned to their corresponding module's pins. After waiting for 40 ns for these lines to be stable, the module is clocked once. The results of the Tree are then scanned out via Scan_Out line to be stored in an output file named multtree_scan_test_result.vectors. This output file (multtree_scan_test_result.vectors) shows the value of the primary inputs and outputs in terms of binary and hexadecimal form for each test set. The process of reading test data from test file, and its subsequent procedures are repeated until all test data have been read. The fault simulation report generated by the VTST Fault Grader reveals that the test patterns provide a 100% fault coverage for the Multiplier Tree module. For details, refer to VTST fault simulation report in the following pages.

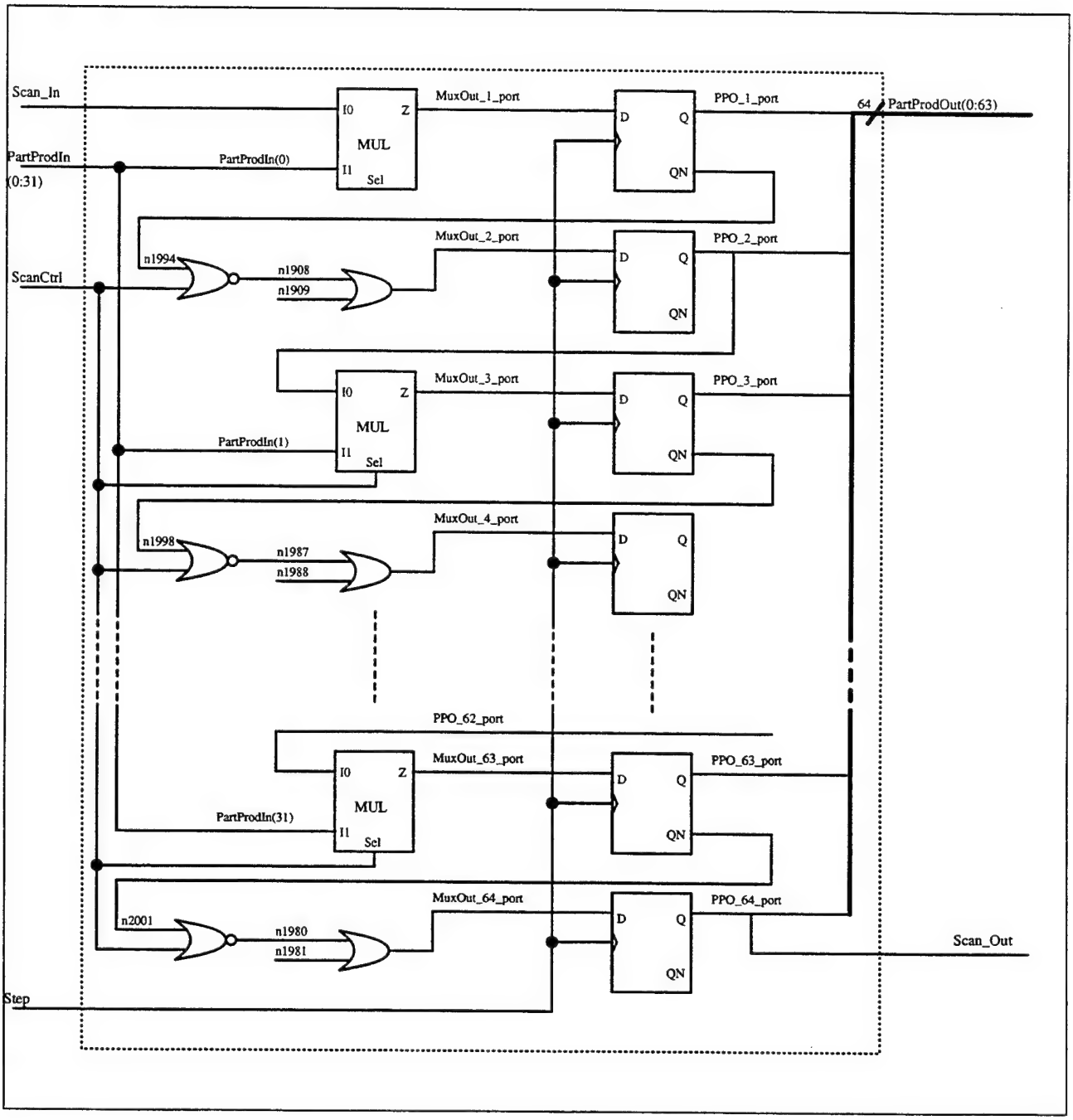


Figure 16.5: Depth 32 Multiplier Stage

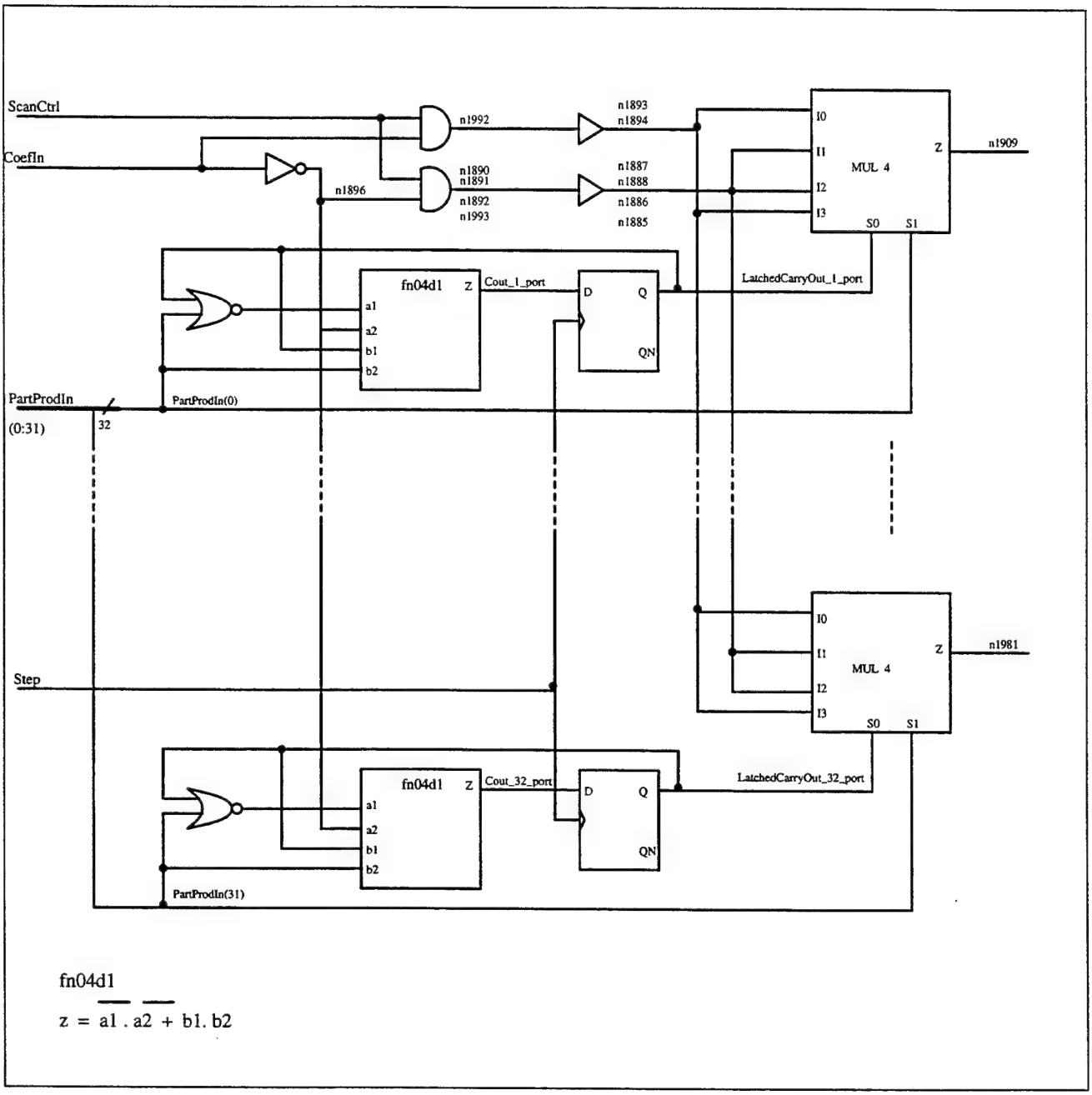


Figure 16.6: Depth 32 Multiplier Stage - continue

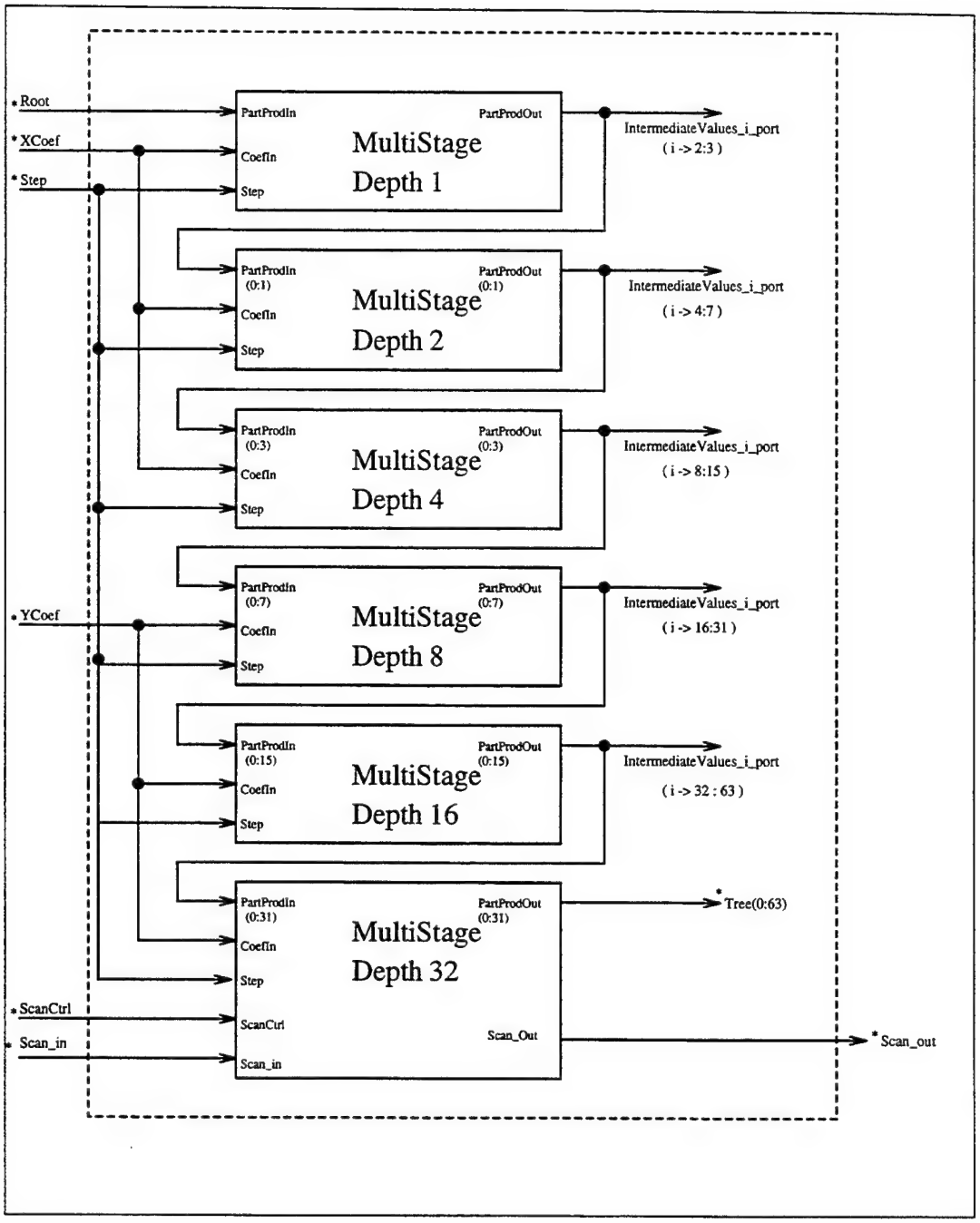


Figure 16.7: Layout of the whole Multiplier Tree

```
************************************************************
*                                                          *
*          VLSI Testability Synthesis Tool (VTST)          *
*            BISTDaTA Tool:Fault Grader-FSIM               *
*                                                          *
*                                                          *
*                 Tue May 21 14:46:14 1996                 *
*                                                          *
************************************************************

This file: multtree_scissor_fsim.rpt

1. Circuit under test : Multiplier Tree

   Number of primary inputs                    = 3
   Number of control inputs                    = 2
   Number of primary outputs                   = 64
   Number of Flip-flops                        = 189
   Number of Latches                           = 0
   Number of Inverter                          = 11
   Number of Tristate buffers                  = 0
   Number of other instances                   = 376
   (Not include PIs,POs,Flip-flops,Latches,and Tristate buffers)

   Depth of the circuit                        = 3

2. Summary of Fault Simulation Results

   Test vectors generated                      = Random generator
   Initial random number generator seed        = 1
   Number of test patterns                     = 65
   Fault coverage                              = 100.00%

   Number of collapsed faults                  = 2288
   Number of detected faults                   = 2288
   Number of undetected faults                 = 0
   Number of faults exist at control lines     = 2
   Total CPU time                              = 2.217 secs

3. Final Signature
   01000100010001101101000010010100100101010001101100000000010100101

3a. Final Hex Signature
    2226b0929a8d005a
```

16.4 ALU Block Description and Module Testing

The following sections give a brief description of the architecture of the ALU Block and its testing at the module level.

16.4.1 ALU Block Description

Figure 16.8 shows the block diagram of this ALUARRAY. It is made of 64 identical ALU blocks. The data inputs to this ALUARRAY are 64 bits Tree from the Multiplier Tree module and 64 bits RdData from the Memory Plane. The output data from this ALUARRAY are the 64 bits WrData connected to the Memory inputs. The writing of these WrData into the Memory Plane is controlled by the WriteEnable lines which must be low for writing to the Memory Plane and high for reading from memory. There is also a scan chain built in this module. Similar to the Multiplier Tree, this scan chain also serves two purposes. The first is to scan out the results from ALUARRAY. The second purpose is to scan test data into the ALUARRAY. The Control signals AControl, BControl, CControl and ALU Control determine the different operations performed in the ALU block.

Figure 16.9 shows the structure of the ALU block. The heart of the ALU block is a full adder with AIn, BIn and CIn as inputs. A multiplexing hardware controls what drives the inputs to the adder. The AIn input is controlled by the signals ACmp, AGetsTreeI, and AGetsSumL according to the truth table shown in Figure 16.10(a). The input to AIn can be selected from Tree, previous sum, forced constant, or its complement. The BIn input is controlled by the signals BCmp, BGetsMemL, and BGetsEnL according to the truth table shown in Figure 16.10(b). The input to BIn can be selected from memory, status of the enable flip-flop, a forced constant, or its complement. The CIn input is controlled by CCmp and CGetsCarry as shown in Figure 16.10(c). The input to this CIn can be selected from the previous value of the carry, a forced constant, or its complement.

Three flip-flops are included in each ALU block. Two are used to store the adder's carryout. They are a carry flip-flop and an enable flip-flop. The Carryout signal is latched into them by the signals LdCarry and LdEnable respectively. The adder's sum which can be written to memory is temporarily stored into the sum flip-flop. A multiplexer and the sum flip-flop make up a scan chain. When ScanCtrl signal is low, no scan in or scan out can be conducted. The ouput of the multiplexer is the adder's sum that can be latched into the sum flip-flop by ALUClock. When ScanCtrl is high, scanning in and scanning out of data via Scan_In and Scan_Out lines can be done. The WriteEnable line is used to enable writing into or reading data from the memory plane. A low in this line will enable writing of data into memory while a high will enable reading of data from the memory. This WriteEnable line is controlled by the ForceEnable, MemWrite and Enable signals.

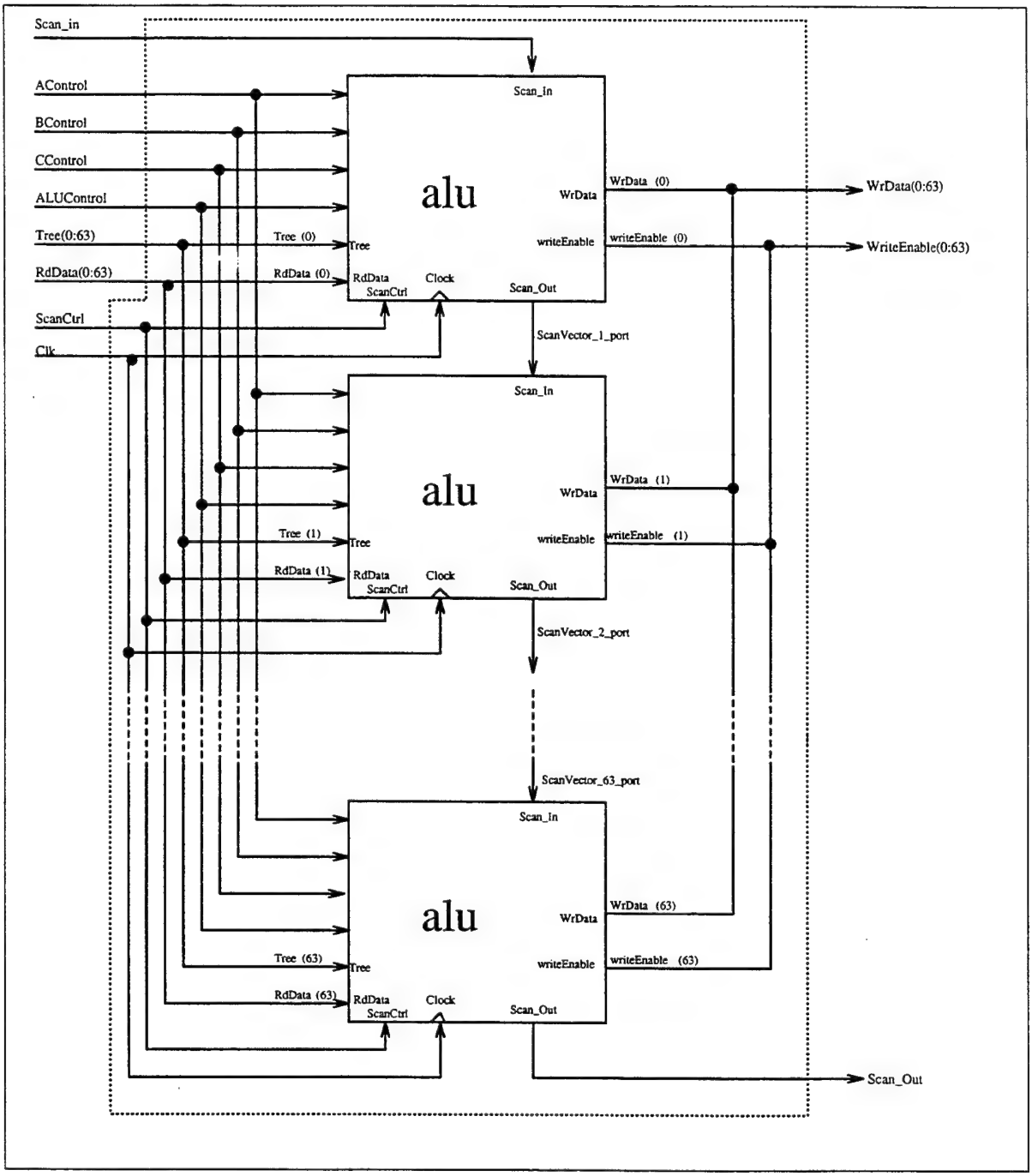


Figure 16.8: Layout of the whole ALU Block

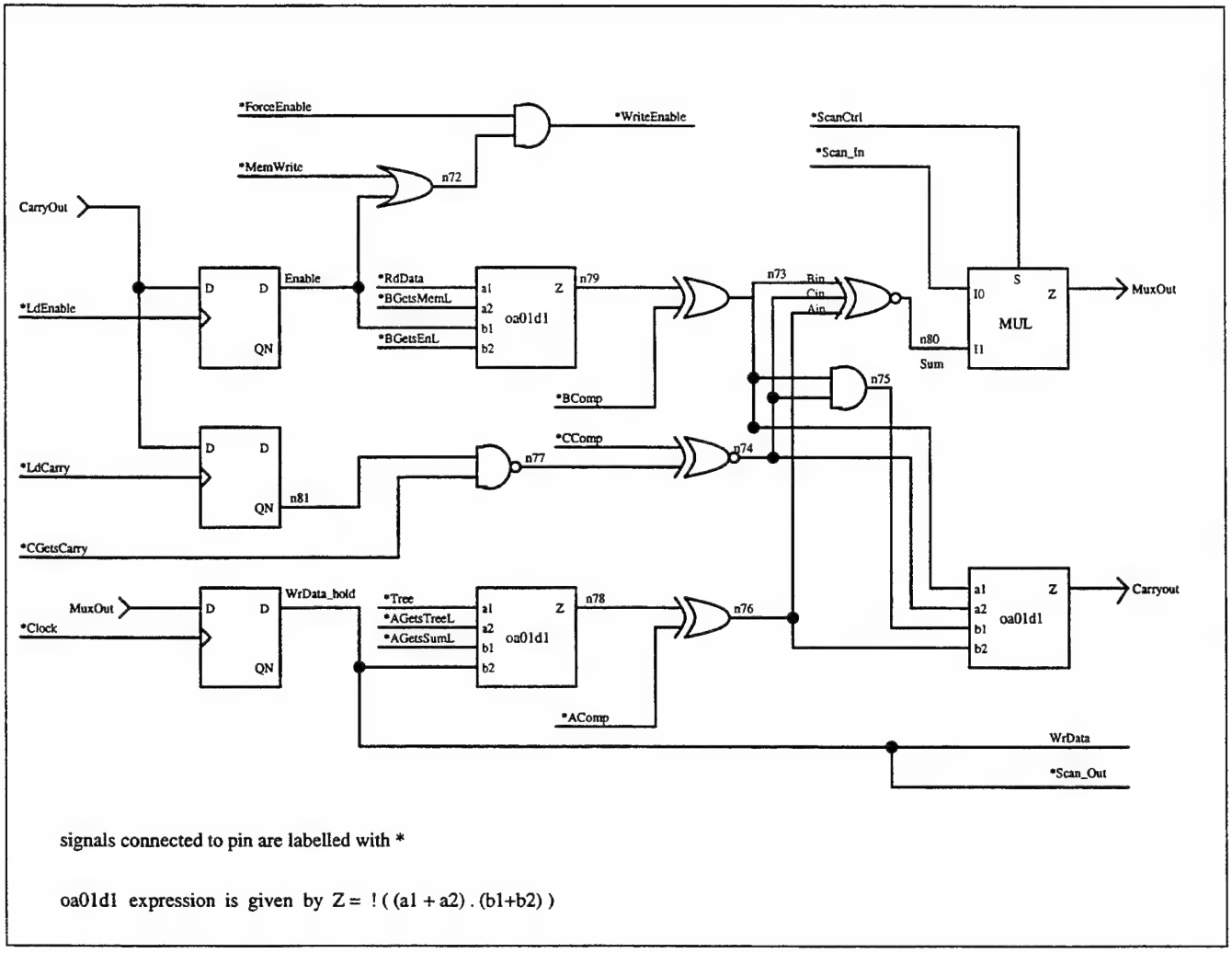
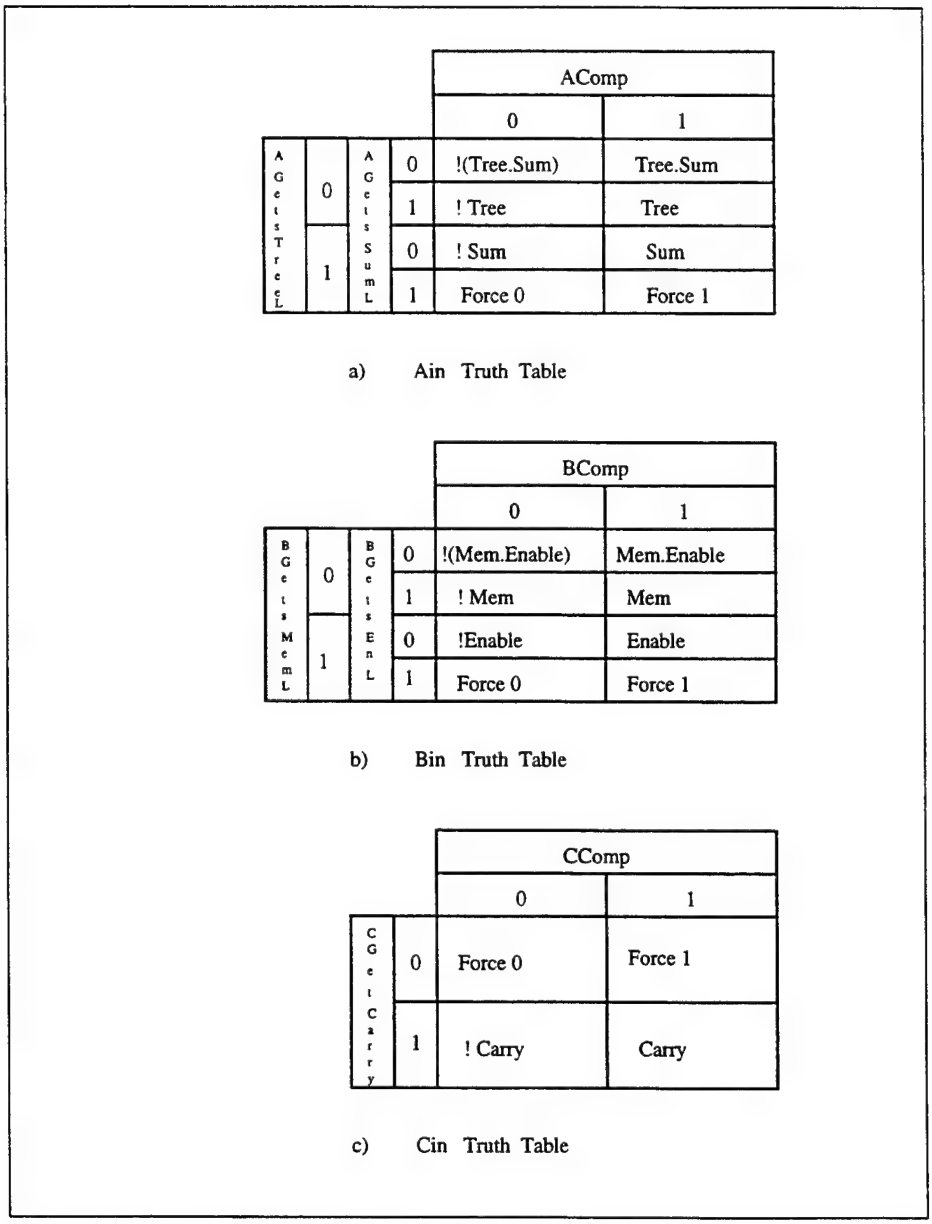


Figure 16.9: Layout of the small ALU block



AGetsTreeL	AGetsSumL	AComp 0	AComp 1
0	0	!(Tree.Sum)	Tree.Sum
	1	! Tree	Tree
1	0	! Sum	Sum
	1	Force 0	Force 1

a) Ain Truth Table

BGetsMemL	BGetsEnL	BComp 0	BComp 1
0	0	!(Mem.Enable)	Mem.Enable
	1	! Mem	Mem
1	0	!Enable	Enable
	1	Force 0	Force 1

b) Bin Truth Table

CGetCarry	CComp 0	CComp 1
0	Force 0	Force 1
1	! Carry	Carry

c) Cin Truth Table

Figure 16.10: Truth Table For AIn, BIn and CIn

16.4.2 ALU Block Module Testing

A VHDL program named test_aluarray_struct_scan.vhd is used to test the whole ALU-ARRAY Block. The program starts off the reading of primary input data from aluarray_test_scan.vectors. There are altogether 100 test data generated by the VTST Test generation program. The sequence of each set of these input data follows that of aluarray port entity declaration starting with ACmp, Tree, AGetsTreeL, AGetsSumL, BCmp, BGetsMemL, BGetsEnL, CCmp, CGetsCarry, RdData, LdCarry, LdEnable, ForceEnable and MemWrite. Having read the primary input data from the test file, the program forces the data at the WrData_hold lines to be the same as RdData. This makes the data at these lines known and not in an ambiguous state. With RdData, Tree and all primary input signals at their respective position, the results of Sum,Carryout and WriteEnable are produced. The results of the Sum and WriteEnable are stored in an output file named aluarray_test_scan_result.vectors. The results of the Sum and Carryout are feedback and stored in three flip-flops for the next computation. The whole process is repeated until the last data set in the test file is applied to the ALU. The output file contains the value of primary input test data in binary form and the outputs (Sum and WriteEnable) produced in binary and hexadecimal forms. The fault simulation report generated by the VTST fault grader tool reveals that the test patterns used to test the ALU array provide a fault coverage of 99.8%. The number of undetected faults is 8 out of 4500 collapsed faults. These undetected faults are testing on the MemWrite line. They would be covered in the testing of Memory Plane at the chip level. For details, refer to the fault simulation report in the following page.

```
************************************************************
*                                                          *
*          VLSI Testability Synthesis Tool (VTST)          *
*            BISTDaTA Tool:Fault Grader-FSIM               *
*                                                          *
*                                                          *
*                 Mon May  6 18:09:33 1996                 *
*                                                          *
************************************************************

This file: aluarray2.ppscan_fbcfree.mod2_fsim.rpt

1. Circuit under test : aluarray

   Number of primary inputs                    = 202
   Number of control inputs                    = 1
   Number of primary outputs                   = 128
   Number of Flip-flops                        = 128
   Number of Latches                           = 0
   Number of Inverter                          = 0
   Number of Tristate buffers                  = 0
   Number of other instances                   = 1217
   (Not include PIs,POs,Flip-flops,Latches,and Tristate buffers)

   Depth of the circuit                        = 5

2. Summary of Fault Simulation Results

   Test vectors generated                      = Pattern file
   Number of test patterns                     = 122
   Fault coverage                              = 99.82%

   Number of collapsed faults                  = 4500
   Number of detected faults                   = 4492
   Number of undetected faults                 = 8
   Number of faults exist at control lines     = 130
   Total CPU time                              = 4.600 secs

3. Final Signature
   11101000101010111110010010011111110110110001111110101100000011010
   11110101111110001101101000110011100001111110111011100110011011111

3a. Final Hex Signature
    715d729fd6cfa185faf1b5cc1e7776af
```

16.5 Memory Plane, Controller Description and Module Testing

The following sections give a description of the memory plane, memory out controller, and the word decoder. Next will be the testing of the memory plane and memory out controller at the module level.

16.5.1 Memory Out Controller and Word Decoder Description

The analysis of this module is based on its VHDL structural description. The VHDL file is named memoryoutcontrol_struct_scan.vhd. Figure 16.11 shows the block diagram of the Memory Out Controller which basically consists of a 6-bit counter (Counter_OutputWordwidth6) and a 6-bit to 64-bit decoder. The counter is enabled when the MemControl Idle line is low. A high in the MemControlReset_Syn will reset the counter to zero when a MemControlClock is applied. Continuing applying the MemControlClock with MemControlReset_Syn pulled low will cause the counter to count up till "111111" and then wrap around to "000000". The outputs of the counter are fed to the decoder and also to the output of the chip as IOAddr.

The inputs of the decoder are fed from the counter outputs. The function of the decoder is to decode the 6-bit code to produce a 64-bit output. Only one of these output bits will be high at a time. Take an example, when the decoder's input is "000000", all the 64 bit outputs will be low except the least significant bit 0. Similarly, a "111111" at its inputs will only produce a high at the most significant output bit of the decoder. These 64 bits from the decoder are fed to the Memory Plane to select a memory cell as an output to the chip's input/output port.

The Word Decoder module is the same as the decoder in the Memory Out Controller. It takes in 6-bit WordAddress signal and decodes it to 64-bit control signal to the Memory Plane. These signals will select a memory cell in the Memory Plane to be written or read by the ALU Block.

16.5.2 Testing of Memory Out Controller

The Memory Out Controller is tested to confirm analysis as described above. A VHDL program named test_memorycontrol_struct_scan.vhd is used to run the test. This program will test solely the Memory Out Controller (test_memorycontrol_struct_scan.vhd). The test program starts with resetting the 6 bit-counter to zero. Then the output of the decoder (IORdEn) and counter (IOAddr) are monitored. The output IOAddr is in hexadecimal form and the IORdEn is in 64-bit binary form. With MemControlIdle and MemControl-

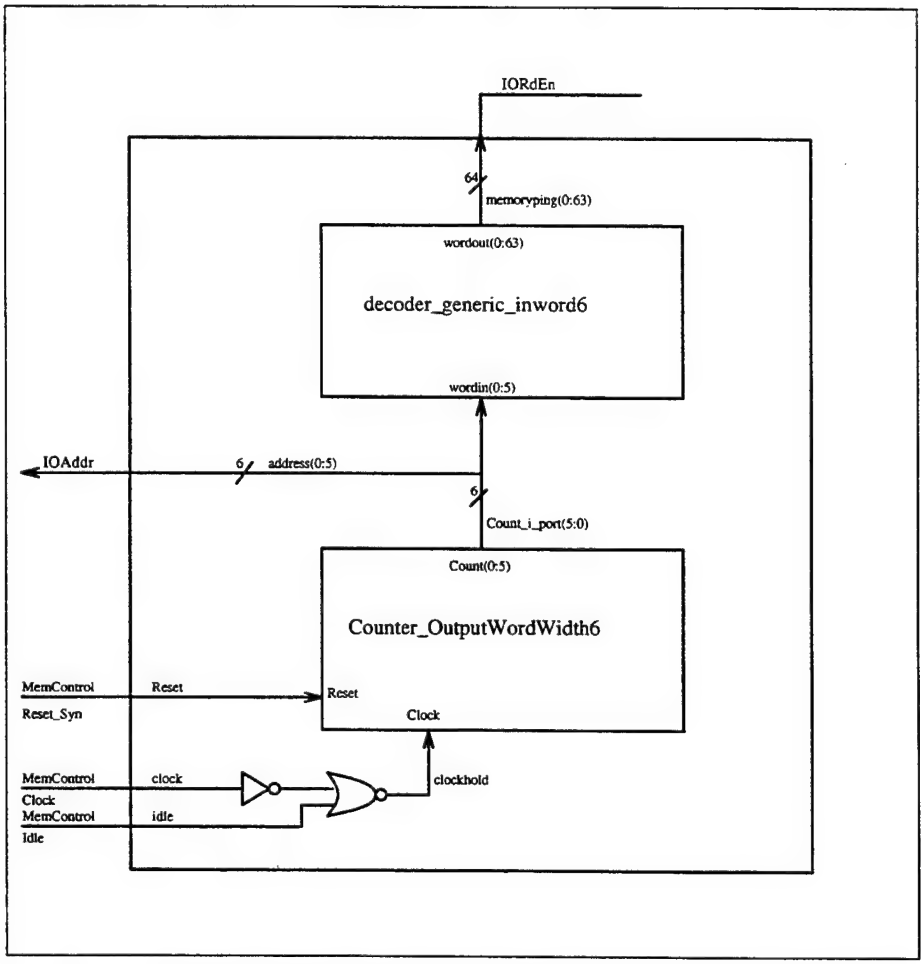


Figure 16.11: Block Diagram of Memory Out Controller

Reset_Syn signals at low, the counter will counter up from "000000" to "111111" and, then back to "000000" at each tick of the MemControlClock. The decoder's output, however; displays a row of 64 bits with "1" shifted left from the least significant bit to the most significant bit position and cycles back to the least significant bit position. For details of the test results, refer to the Appendix.

16.5.3 Memory Plane description

The analysis of this Memory Plane module is based on its given VHDL behavioral description. The VHDL file is named pixmem.vhd. This Memory Plane consists of two buffers. Each of buffer has 64 cells, and each cell consists of a 64 bit word. The entity port signals of this module consist of Wordlines, Rowselect, ALUWrData, ALURdEnL, ALURdData, IORdData and IORdEn. The Wordlines signals (64 bits) will determine which buffer cell to be written or read from by the ALU. When the Rowselect line is low, it will connect buffer1 to the ALU and buffer2 to IO. Likewise, when Rowselect line is high, buffer1 will be connected to IO and buffer2 will be connected to ALU. The ALUWrData are 64 bit outputs of the ALU to be written into the memory. The ALURdEnL are 64 bit control lines that will enable ALUWrData to be written into memory only when they are low. The ALURdData are 64 bits output data read from memory to the ALU. The IORdData are 32 bit output data from the memory to the input/output (IO). The IORdEn are 64 bits address lines which select a particular 32 bits to be output to the IO. All the 32 bits of the Memory Plane to the IO are passed through tri_state buffer controlled by a line called IO_OutEn. A high in this line will enable the flow of the 32 bits out of the IO. A low at this line will produce high impedance at the IO lines.

Three operations are performed in the Memory Plane. They are the ALU Write Cycle, ALU Read Cycle and IO Read Cycle. To perform a ALU Write Cycle, all the ALUWrEn lines from the ALU have to be pulled low. The ALUWrData, RowSelect, and Wordlines from the ALU must be set on the rising edge of the MemControlClock. With these lines held for a MemControlClock cycle, the ALUWrData will be written into the appropriate buffer's cell.

As for the ALU Read Cycle, all the ALUWrEn lines have to be pulled high. The RowSelect and Wordlines must be set on the rising edge of the MemControlClock. ALURdData will be valid on the rising edge of the next MemControlClock cycle. These ALURdData will be fed back to the ALU.

As for the IO Read Cycle, the IORdEn from the Memory Out Controller must be set to an appropriate value on the rising edge of MemControlClock. Similarly, the IO_OutEn line has to be high also. The IORdData will be valid on the rising edge of the next MemControlClock cycle and these 32 bit IORdData will be available at the outputs of the chip.

16.5.4 Testing of Memory Plane

The idea of generating the test patterns for the Memory Plane is obtained from RAM BIST of LSI Logic Corp. Fault detection during testing of a RAM could fall into two categories : (1) address short, and (2) a struck-at fault of a memory cell. These can be detected by writing a set of patterns into the memory and then read each address separately and check if what is read is what was written. A complete and safe method is to test each cell against another cell to see if any cell is stuck at 0 or stuck at 1, any two cells are shorted together, or any two addresses are shorted as one.

A test file named pixmem_test.vectors containing fourteen sets of test patterns are used to test the Memory Plane. In the first test cycle, all bits in memory are partitioned into two sections to verify that no cell in one section is shorted to any cell in another section. The second test cycle then cuts each of the two halves in two to verify that, for each of the four quarters, all bits in the quarter are independent of bits in the other three quarters. This procedure of bisecting continues until no further partition can be made. A look at the fourteen sets of test patterns reveals that the cuttings are done on address space and data space simultaneously. This saves the number of cycles of doing them separately and obtains the same results. These test patterns cover all cell stuck-at-1/0 faults, state transition faults, address decoder faults and the state coupling faults.

A test program named test_pixmem_v1.vhd is used together with pixmem_test.vectors file to test the Memory Plane. This program will read 128 rows of test patterns in each test cycle from the pixmem_test.vectors file, then write them into every cells in the two buffers of the Memory Plane. After that, all stored data in each cell are read via ALURdData to an output file named pixmem_test_result.vectors. At the same time, IORdData are also read via IORdData lines into an output file named pixmem_IORd_test_result.vectors. The whole process is repeated until all the fourteen cycles of test data are done. The ALURdData and IORdData data in both pixmem_test_result.vectors and pixmem_IORd_test_result.vectors (see Appendix D-4) files are the results produced by the Memory Plane test.

16.6 Testing of Multiplier, ALU Block and Memory Plane at the chip level

The testing of the Multiplier, ALU Block and Memory Plane at chip level is done by a VHDL program named test_plane_overall.vhd. For details, refer to the code listed in Appendix E-1. The program could be divided into three portions. The first portion is to test the Multiplier, the second is to test the ALU Block and the last one is to test the Memory Plane. They are described in the following sections. Test data are generated by the VTST toolset and stored in three test data files to test the three different portions of the chip. The test results produced are stored in four different output files. The verification of these test results is done by comparing them with the test results produced from module level testing. The total number of test clocks required to test the whole chip is shown below:

a) Multtree

No. of test patterns : 65
No. of test pattern : 128
Total no. of clock to test Multtree
= 65 X 128
= 8320

b) ALU

No. of test patterns : 100
No. of test pattern : 128
Total no. of clock to test Multtree
= 100 X 128
= 12800

c) Memory

No. of test cycle : 14
No. of test pattern per cycle :128
No. of clock per test pattern
= 64 (scan in) + 64 (scan out)
= 128 clocks
Total no. of clocks required to test memory
= 14 X 128 X 128
= 229376

d) Total no. of clocks required to test the chip
= 8320 + 12800 + 229376
= 250496
$< 2^{18}$

16.6.1 Testing of Multiplier

A close look at the top level architecture of the EMC chip (see Figure 16.2) reveals that the 64 bits of the Tree output of the Multiplier Tree cannot be reached directly. The only way to access these results is to scan them out via the Scan_Out line.

The testing in this portion only involves seven pins. They are ACoef, BCoef, CCoef, MultiClock, Scan_In, Scan_Out and ScanCtrl. The rest of the pins are set to passive state. Just like the testing of the Multiplier Tree at the module level, this program starts with setting ACoef, BCoef and CCoef to low and clocking the MultiClock thirteen times. This will set all the flip-flops in the Multiplier Tree module to a known state. What follows is the reading of the primary input test datas (ACoef, BCoef and CCoef) from a test file named plane_multtree_test.vectors. These read in test data are then assigned to their prospective pins. The program waits for 40ns for the signals and outputs to be stable before latching them into their internal flip-flops. Having done that, the results are scanned out by 128 clocks. These results flow via the scan chain in Multiplier Tree and ALU

Block, and out of the Scan_Out line. These results are stored in an output file named plane_multtree_test_result.vectors. The reading of test data, computation and outputing of results continue until all the test data in the test file have been tested.

The sets of test data contained in the plane_multtree_test.vectors file are the same as those used in the testing of Multiplier Tree at the module level. The purpose is for the comparison of their output results. The output results contained in plane_multtree_test_result.vectors file show the values of the primary input data in binary form, and output results in both binary and hexadecimal forms. These results can be compared with the results produced by the Multiplier Tree testing at module level. The fault coverage for this test is the same as the fault coverage of the Multiplier Tree module testing. The fault coverage is 100%. For detail, refer to the section on Multiplier Tree testing at module level.

16.6.2 Testing of ALU Block

A look at the top level architecture of the EMC chip (see Figure 16.2) reveals that the following inputs are required to test the ALU Block. They are AControl, BControl, CControl and ALU Control signals, ALURdData and TreeData data bus. These control signals could be applied from the external pins of the chip. However, the ALURdData and TreeData data cannot be reached from external pins. The TreeData can be obtained only by scanning in data via Scan_In pin into the scan chain in the Multiplier Tree to the ALU Block. Similarly, the ALURdData could be obtained only by writing data into the Memory Plane from the ALU Block through TreeData, and then read them back from the Memory Plane into ALURdData lines. By varying the control signals and data, the ALU Block will produce outputs at ALUWrData. These outputs could only be obtained by scanning them out of the ALU Block via Scan_Out line. In order to save the number clock cycle needed, the scanning in of data and scanning out of results are done simultaneously.

The second portion of the test_plane_overall.vhd used to test the ALU Block is described below. The program starts with the initialization of all the control signals. It is then followed by reading a set of test data from a test file named plane_alu_test.vectors. The sequence of each set of these input data follows that of aluarray port entity declaration starting with ACmp, TreeData, AGetsTreeL, AGetsSumL, BCmp, BGetsMemL, BGetsEnL, CCmp, CGetsCarry, ALURdData, LdCarry, LdEnable, ForceEnable and MemWrite. All the control signals can be assigned directly to their prospective pins. With ScanCtrl at low, the RdData and TreeData are scanned with 128 clock cycles. The results of the first scan out are not stored as they are not a valid one. Analyzing the architecture of ALU block in Figure 16.9, we notice that the scanned in ALURdData is now only at the WrData_hold position. In order to observe these at RdData, these data need to be written into the Memory Plane and then read back. Having done this, the data at the RdData and the WrData_hold have been set to the same values. With all control signals and data in position, the program will wait for 50ns for the Sum, Carryout to be stable before they are latched into three flip-flops for the next cycle computation. After this, the program will read in the next set of test data and then scan in the data and scan out

the results. This second set of results is then valid and stored in the output file named plane_alu_test_result.vectors. The process will continue until all sets of test data in the test file have been tested.

The sets of test data contained in the plane_alu_test.vectors file are the same as those used in the testing of ALU Block at module level. The purpose is for the comparison of their output results. The output results contained in plane_alu_test_result.vectors file show the values of the primary input data in binary form, and output results in both binary and hexadecimal forms. These results can be compared with the results produced by ALU testing at the module level. The fault coverage for this test is the same as the fault coverage of the ALU array module testing. The fault coverage is 99.8%. For detail, refer to the section on ALU array testing at module level.

16.6.3 Testing of Memory Plane

The testing of the Memory Plane includes the testing of the address and data spaces in the Memory Plane. The test also includes the testing of the ForceEnable, MemWrite and Enable lines of the ALU Block. The test also reveals the functionality of the Memory Out Controller, Word Decoder and IORdData. As the writing of the test patterns cannot be done directly through the external pins, they are scanned in via Scan_In pin to ALU Block and then written into the Memory Plane. The test patterns written into the Memory Plane are then scanned out through the ALU Block via Scan_Out line. The IORdData from the Memory Plane, however; can be read directly from the external IORdData lines.

The other portion of the test_plane_overall.vhd used to test the Memory Plane is described below. The program starts with the initialization of all the control signals. It is then followed by reading in a set of test data from a test file named plane_memory_test_v1.vectors. This program will read the first 64 rows of test patterns in each test cycle from the plane_memory_test_v1.vectors file, then write them into every cell of the first buffers of the Memory Plane. After that, the program wil set MemWrite to high, read the second 64 rows of test patterns of each test cycle and then write them into all the cells of the second buffer of the Memory Plane. This sector of the program tests the functionality of the MemWrite line. Following this, all the stored data in each cell are read via ALURdData, through ALU Block and then scanned out. The scanned out results are then stored in an output file named plane_memory_test_result.vectors. At the same time, IORdData are also read via IORdData lines into an output file named plane_memory_IORd_test_result.vectors. The whole process is repeated until all fourteen cycles of test data are done. The ALURdData and IORdData data in both plane_memory_test_result_v1.vectors and plane_memory_IORd_test_result.vectors files are the output results produced by the Memory Plane test.

The set of test data contained in the plane_memory_test_v1.vectors file are the same as those used in the testing of the Memory Plane at the module level. The purpose is for the comparison of their output results. The fault coverage for this test is the same as the fault coverage of the Memory Plane module testing. The fault coverage is 100%. For detail,

refer to the section on Multiplier Tree at the module level. Beside this, the testing done here also includes the testing of MemWrite line that are not in the previous tests.

16.7 Conclusion

This report starts with the analysis and testing of each main component in the Enhanced Memory Chip at the module level. The anlysis and testing of the same components are done later at the chip level. The test data used to perform the testing are generated by the VTST test generation program. The purpose of doing the testing of components at two different level is to use the results of the testing at module level to verify the correctness of the tests at the chip level. Finally, the VTST fault simulation report reveals that the test patterns used to perform the testing provide a 100% fault coverage.

16.8 Appendix: VTST Experimental Results

```
*****************************************************************
*       circuit under test:     multtree                        *
*       input circuit file:     multree_structural.vhd          *
*       technology lib.   :     compass library (vsc450.tech)   *
*****************************************************************

1. Case 1: with only circuit preprocessor (SCISSOR) and insertion of BIST
        test hardwares (TPG, MISR, AND tree)

2.1 Circuit analysis

   no. of PIs:  5 (including control signals such as clock)
   no. of POs:  64
   no. of DFFs: 189
   no. of gates: 386 (other than DFFs)

2.2 Summary of Fault Simulation Results

   Test vectors generated                    = LFSR
   Initial seed                              = 1
   Number of test patterns                   = 1024
   Fault coverage                            = 97.64%

   Number of collapsed faults                = 2288
   Number of detected faults                 = 2234
   Number of undetected faults               = 54
   Number of faults exist at control lines   = 2
```

```
   Total CPU time                                  = 11.267 secs

2.3 Final Signature
   00110110001100101111110111001110100011001111011001100000101011000

   Final Hex Signature
   c6c4fb3713f668a1

2.4 BIST overhead analysis

        GATE COUNT BEFORE MODIFY : 16605
        GATE COUNT ATFER MODIFY : 18018
        OVERHEAD is : 0.078422
        GATE COUNT for TPG (3-bit) : 488
        GATE COUNT for MISR (64-bit) : 11112
        GATE COUNT for all XOR arrays :  0
        GATE COUNT for TEST CONTROL LOGICS : 171
        GATE COUNT TOTAL : 29789
        OVERHEAD is : 0.442579


2. case 2: with circuit preprocessor (SCISSOR), Circular BIST methodology,
           and test hardwares.

2.1 Circuit analysis


        Primary Inputs (PIs)            : 3
        Control Inputs (CPIs)           : 2
        Primary Outputs (POs)           : 64
        Latches                 : 0
        Flip flops                      : 189
        Other Instances (except PIs, POs, FFs, Latches): 512
        Number of Flip flops in feedback path:68

2.2 Summary of Fault Simulation Results

   Test vectors generated                       = LFSR
   Initial seed                                 = 1
   Number of test patterns                      = 1024
   Fault coverage                               = 99.82%

   Number of collapsed faults                   = 2792
   Number of detected faults                    = 2787
   Number of undetected faults                  = 5
   Number of faults exist at control lines      = 2
   Total CPU time                               = 9.150 secs

2.3 Final Signature
```

```
   01110111100010011010110011100000011111001001101110001111010001 01

   Final Hex Signature
   ee195370e39d1f2a

2.4 BIST overhead analysis

GATE COUNT BEFORE MODIFY : 16605
GATE COUNT ATFER MODIFY : 21924
OVERHEAD is : 0.242611
GATE COUNT for TPG (3-bit) : 488
GATE COUNT for MISR (64-bit) : 11112
GATE COUNT for all XOR arrays :  0
GATE COUNT for TEST CONTROL LOGICS : 171
GATE COUNT TOTAL : 33695
OVERHEAD is : 0.507197


*****************************************************************
*       circuit under test:     aluarray (4 slice ALU)          *
*       input circuit file:     aluarray_structural.vhd          *
*       technology lib.    :    compass library (vsc450.tech)   *
*****************************************************************

1. Case 1: with only circuit preprocessor (SCISSOR) and insertion of BIST
        test hardwares (TPG, MISR)

1.1 Circuit analysis

Primary Inputs (PIs)            : 21 (including control signals such as clock)
Primary Outputs (POs)           : 8
Latches                         : 0
Flip flops                      : 12
Other Instances (except PIs, POs, FFs, Latches: 72
Number of Flip flops in feedback path:12

1.2 Summary of Fault Simulation Results

   Test vectors generated                       = LFSR
   Initial seed                                 = 1
   Number of test patterns                      = 103
   Fault coverage                               = 100.00%

   Number of collapsed faults                   = 316
   Number of detected faults                    = 316
   Number of undetected faults                  = 0
   Number of faults exist at control lines      = 2
   Total CPU time                               = 0.133 secs
```

```
1.3 Final Signature
   01001011

   Final Hex Signature
   2d

1.4 BIST overhead analysis

        GATE COUNT BEFORE MODIFY : 1736
        GATE COUNT ATFER MODIFY : 1880
        OVERHEAD is : 0.076596
        GATE COUNT for TPG (18 bit) : 2537
        GATE COUNT for MISR (8 bit) : 1499
        GATE COUNT for all XOR arrays :  0
        GATE COUNT for TEST CONTROL LOGICS : 545
        GATE COUNT TOTAL : 6461
        OVERHEAD is : 0.731311

2. Case 2: with circuit preprocessor(SCISSOR), Pseudo-Scan(SCISSOR-feedback-
        free), Signal reduction (BISTSYN) and BIST test hardwares
        (TPG and MISr).

2.1 Circuit analysis

        Primary Inputs (PIs)            : 33
        Control Inputs (CPIs)           : 0
        Primary Outputs (POs)           : 4
        Latches                 : 0
        Flip flops                      : 0
        Other Instances (except PIs, POs, FFs, Latches: 72

2.2 Summary of Fault Simulation Results

   Test vectors generated                       = LFSR
   Initial seed                                 = 1
   Number of test patterns                      = 30
   Fault coverage                               = 100.00%

   Number of collapsed faults                   = 274
   Number of detected faults                    = 274
   Number of undetected faults                  = 0
   Number of faults exist at control lines      = 16
   Total CPU time                               = 0.050 secs

2.3 Final Signature
   110111101011

   Final Hex Signature
   b7d
```

2.4 BIST overhead analysis

GATE COUNT BEFORE MODIFY : 1736
GATE COUNT ATFER MODIFY : 2232
OVERHEAD is : 0.222222
GATE COUNT for TPG (13 bit) : 1899
GATE COUNT for MISR (12 bit) : 2174
GATE COUNT for all XOR arrays : 0
GATE COUNT for TEST CONTROL LOGICS : 545
GATE COUNT TOTAL : 6850
OVERHEAD is : 0.746569

```
****************************************************************
*       circuit under test:     aluarray (64 slice ALU)        *
*       input circuit file:     aluarray_structural.vhd        *
*       technology lib.    :    compass library (vsc450.tech)  *
****************************************************************
```

1. Case 1: using 141-bit LFSR and 128-bit MISR

1.1 Circuit analysis

area : 27776 units
no. of PIs: 141 (including control signals such as clock)
no. of POs: 128
no. of DFFs: 192
no. of gates: 1152 (other than DFFs)

1.2 Summary of Fault Simulation Results

Test vectors generated	= LFSR
Initial seed	= 1
Number of test patterns	= 376
Fault coverage	= 100.00%
Number of collapsed faults	= 4756
Number of detected faults	= 4756
Number of undetected faults	= 0
Number of faults exist at control lines	= 6
Total CPU time	= 6.000 secs

1.3 Final Signature

00100101101110110110000100111101001110000011010111101100110000100100001000001
00111100100010010001000100100111000100001100000110O

Final Hex Signature

4add68cbc1ca73342409722119c11603

1.4 BIST overhead analysis

GATE COUNT BEFORE MODIFY : 27776
GATE COUNT ATFER MODIFY : 36032
OVERHEAD is : 0.229130
GATE COUNT for TPG : 19168
GATE COUNT for MISR : 11112
GATE COUNT for all XOR arrays : 1408
GATE COUNT for TEST CONTROL LOGICS : 3185
GATE COUNT TOTAL : 70905
OVERHEAD is : 0.608265

2. case 2: with only circuit preprocessor (SCISSOR) and insertion of BIST test hardwares (TPG, MISR, XOR tree, AND tree)

2.1 Circuit analysis

no. of PIs:	138
no. of CPIs:	4
no. of POs:	64
no. of DFFs:	192
no. of gates:	1152 (other than DFFs)
size of TPG:	138
size of MISR:	64

2.2 Summary of Fault Simulation Results

Test vectors generated	= LFSR
Initial seed	= 1
Number of test patterns	= 375
Fault coverage	= 100.00%
Number of collapsed faults	= 4884
Number of detected faults	= 4884
Number of undetected faults	= 0
Number of faults exist at control lines	= 2
Total CPU time	= 5.233 secs

2.3 Final Signature
10110100100101010101101100110101110101110010010000011101000010000

Final Hex Signature
d29aadcabe428b01

2.4 BIST overhead analysis

GATE COUNT BEFORE MODIFY : 27776
GATE COUNT ATFER MODIFY : 30080
OVERHEAD is : 0.076596
GATE COUNT for TPG : 19168
GATE COUNT for MISR : 11112
GATE COUNT for all XOR arrays : 1408
GATE COUNT for TEST CONTROL LOGICS : 3185
GATE COUNT TOTAL : 64953
OVERHEAD is : 0.572368

3. case 3: with circuit preprocessor(SCISSOR), Pseudo-Scan(SCISSOR-feedback-free), Signal reduction (BISTSYN) and BIST test hardwares.

3.1 Circuit analysis

no. of PIs: 138
no. of CPIs: 4
no. of Pseudo-PIs: 192
no. of POs: 64
no. of Pseudo-POs:128
no. of DFFs: 192
no. of gates: 1152
size of TPG: 13
size of MISR: 64

3.2 Summary of Fault Simulation Results

Test vectors generated	= LFSR
Initial seed	= 1
Number of test patterns	= 1024
Fault coverage	= 97.75%
Number of collapsed faults	= 3994
Number of detected faults	= 3904
Number of undetected faults	= 90
Number of faults exist at control lines	= 0
Total CPU time	= 32.550 secs

3.3 Final Signature

00000000000000000000000000000000000000000000000000000000000000000000

Final Hex Signature
0000000000000000

3.4 BIST overhead analysis

GATE COUNT BEFORE MODIFY : 27776
GATE COUNT ATFER MODIFY : 35712
OVERHEAD is : 0.222222

```
GATE COUNT for TPG : 1899
GATE COUNT for MISR : 11112
GATE COUNT for all XOR arrays :  2816
GATE COUNT for TEST CONTROL LOGICS : 3185
GATE COUNT TOTAL : 54724
OVERHEAD is : 0.492435
```

4. case 4: with circuit preprocessor (SCISSOR), Circular BIST methodology, and test hardwares.

4.1 Circuit analysis

```
no. of PIs:     138
no. of CPIs:    4
no. of POs:     128
no. of DFFs:    192
no. of gates:   1344 (including XORs added)
size of TPG:    138
size of MISR:   64
```

4.2 Summary of Fault Simulation Results

```
Test vectors generated                      = LFSR
Initial seed                                = 1
Number of test patterns                     = 287
Fault coverage                              = 100.00%

Number of collapsed faults                  = 5780
Number of detected faults                   = 5780
Number of undetected faults                 = 0
Number of faults exist at control lines     = 2
Total CPU time                              = 4.450 secs
```

4.3 Final Signature

```
01110100010111001100101111011110111111000010110010001101000001 01
```

```
Final Hex Signature
e2a33db7f3431b0a
```

4.4 BIST overhead analysis

```
GATE COUNT BEFORE MODIFY : 27776
GATE COUNT ATFER MODIFY : 36032
OVERHEAD is : 0.229130
GATE COUNT for TPG : 19168
GATE COUNT for MISR : 11112
```

```
GATE COUNT for all XOR arrays :  1408
GATE COUNT for TEST CONTROL LOGICS : 3185
GATE COUNT TOTAL : 70905
OVERHEAD is : 0.608265
```

Bibliography

[1] T. W. Williams and K. P. Parker, "Design for Testability - A survey," IEEE Trans. on Computers, vol. C-31, pp. 2-15, Jan. 1982.

[2] E. J. McCluskey, "Built-In Self Test Techniques", IEEE Design and Test of Computers, pp. 21-28, Apr. 1985.

[3] V. K. Agrawal and E. Cerney, "Store and Generate Built-In Testing Approach", Proc. 11th Int. Symp. Fault-Tolerant Computing, pp. 35-40, 1981.

[4] D. Komonystky, "LSI Self-Test Using LSSD and Signature Analysis", Proc. Int. Test Conf., pp. 414-424, 1982.

[5] Y. M. El-Zig, "S_3: VLSI Self-Testing Using A Signature Analysis and Scan-Path Techniques", Proc. Int. Conf. Computer-Aided Design, pp. 73-76, 1983.

[6] P. H. Bardell and W. H. McAnney, "Self-Testing of Multichip Logic Modules", Proc. Int. Test Conf., pp. 200-204, 1982.

[7] P. P. Fasang, "BIDCO, Built-In Digital Circuit Observer", Proc. Int. Test Conf., pp. 261-266, 1980.

[8] A. Kransniewski and A. Albicki, "Automatic Design of Exhaustively Self-Testing Chips with BIBLO modules", Proc. Int. Test Conf., pp. 362-371, 1985.

[9] B. Koenemann et al., "Built-In Logic Block Observation Techniques", Proc. Int. Test Conf., pp. 37-41, 1979.

[10] Z. Barzilai, J. Savir, G. Markowsky, and M. G. Smith, "The Weighted Syndrome Sums Approach to VLSI Testing", IEEE Trans. on Computers, vol. C-29, pp. 1012-1013, Nov. 1981.

[11] E. J. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique", IEEE Trans. on Computers, vol. C-33, No. 6, June 1984.

[12] S. B. Akers, "On the Use of Linear Sums in Exhaustive Testing", Proc. 15th Fault Tolerant Comp. Symp., pp. 148-153, June 1985.

[13] N. Vasanthavada and P. N. Marinos, "An Operationally Efficient Scheme for Exhaustive Test-Pattern Generation Using Linear Codes", Proc. Int. Test Conf., pp. 476-482, 1985.

[14] L. T. Wang and E. J. McCluskey, "Condensed Linear Feedback Shifter Register (LFSR) Testing - A Pseudoexhaustive Test Technique", IEEE Trans. on Computers, vol. C-35, No. 4, 1986.

[15] L. T. Wang and E. J. McCluskey, "Circuits for Pseudo-exhaustive Test Pattern Generation", Proc. Int. Test Conf., pp. 25-37, 1986.

[16] W. W. Peterson and E. J. Weldon, Jr., Error Correcting Codes, 2nd Edition, M.I.T. Press, 1972.

[17] C.-I. H. Chen and J. Yuen, "Autonomous - Tool for Hardware Partitioning in a Built-In Self-Test Environment," Proc. of IEEE International conference on Computer Design, pp. 264-267, 1992.

[18] F. Brglez, P. Pownall and R. Hum, "Accelerated ATPG and Fault Grading Via Testability Analysis," Proc. IEEE Int. Symp. Circuits and Systems, pp. 695-698, 1985.

[19] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," IEEE Trans. on Computers, vol. C-32, pp. 1137-1144, 1983.

[20] W.-T. Cheng and M.-L. Yu, "Differential Fault Simulation - A Fast Method Using Minimal Memory," Proc. of 26th Design Automation Conference, pp. 424-428, 1989.

[21] W.-T. Cheng and T. J. Lewandowski, "GENTEST, an Automatic Test Generation System for Sequential Circuits," Computer Magazine, April, 1989.

[22] S. B. Akers and W. Jansz, "Test Set Embedding in a Built-In Self-Test Environment," Proc. International Test Conference, pp. 257-263, 1989.

[23] R. D. Dixon, M. Calle, C. W. Longway, L. Peterson and R. Siferd, "The SF1 real Time Computer," Proc. of IEEE National Aerospace and Electronics Conference, pp. 60-64, 1988.

[24] G. Kane, MIPS RISC Architecture, Prentice Hall, 1988.

[25] E. Reingold, J. Nievergelt and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, pp. 353-354, 1977.

[26] E. Trischler, "Incomplete Scan Path with an Automatic Test Generation Methodology". *Proc. International Test Conference*, pages 153–162., 1980.

[27] V. D. Agrawal, K. T. Cheng, D. D. Johnson, and T. Lin, "Designing Circuits with Partial Scan". *IEEE Design Test Comput.*, 5:8–15., 1988.

[28] D. H. Lee and S. M. Reddy, "On Determining Scan Flip-Flops in Partial Scan Designs". *Proc. Int'l Conf. on Computer-Aided Design*, pages 322–325, November 1990.

[29] V. Chickermane and J. H. Patel, "An Optimization Based Approach to the Partial Scan Problem". *Proc. Int'l Test Conf.*, pages 377–386, September 1990.

[30] K. T. Cheng and V. D. Agrawal, "A Partial Scan Method for Sequential Circuits with Feedback". *IEEE Trans. Computer*, 39(4):544–548, April 1990.

[31] M. Abramovici, K. B. Rajan, and D. T. Miller, "FREEZE!: A New Approach for Testing Sequential Circuits". *29th ACM/IEEE Design Automation Conf.*, pages 22–25, 1992.

[32] A. Krasniewski and S. Pilarski, "Circular Self-Test Path: A Low-Cost BIST Technique for VLSI Circuits". *IEEE Trans. Computer-Aided Design*, 8(1):46–55, January 1989.

[33] LSI Logic, Milpitas, CA. *" 1.0 Micron Cell-Based Products Databook"*, February 1991.

[34] W. T. Cheng and M. L. Yu, "Differential Fault Simulation - A Fast Method Using Minimal Memory". *26th ACM/IEEE Design Automation Conf.*, pages 424–428, 1989.

[35] T. M. Niermann, W. T. Cheng, and J. H. Patel, "PROOFS: A Fast, Memory Efficient Sequential Circuit Fault Simulator". *27th ACM/IEEE Design Automation Conf.*, pages 535–540, 1990.

[36] C.-I. H. Chen and G. Sobelman, "An Efficient Approach To Pseudo-Exhaustive Test Generation For BIST Design," Proc. of IEEE International Conference on Computer Design, pp. 576-579, 1989.

[37] C. L. Chen, "Exhaustive Test Pattern Generation Using Cyclic Codes," IEEE Trans. on Computers, vol. C-37, No. 2, pp. 225-228, 1988.

[38] P. Golan, O. Novak and J. Hlavicka, "Pseudoexhaustive Test Pattern Generator with Enhanced Fault Coverage," IEEE Trans. on Computers, vol. C-37, No. 4, 1988.

[39] S. Hellebrand, H.-J. Wunderlich and O. F. Haberl, "Generating Pseudo-Exhaustive Vectors for External Testing," Proc. 1990 IEEE Int. Test Conf., pp. 670-679, 1990.

[40] E. J. McCluskey and S. Bozorgui-Nesbat, "Design for Autonomous Test," IEEE Trans. on Circuits and Systems, vol. CAS-28, No. 11, pp. 1070-1078, 1981.

[41] O. Patashnik, "Circuit Segmentation for Pseudo Exhaustive Testing," Center for Reliable Computing Tech. Report No: 83-14, Stanford Univ., Oct., 1983.

[42] E. C. Archambeau, "Network Segmentation for Pseudo-Exhaustive Testing," Center for Reliable Computing Tech. Report No: 85-10, Stanford Univ., July, 1985.

[43] M. W. Roberts and P. K. Lala, "An Algorithm for the partitioning of Logic Circuits," IEE Proc. pp. 113-118, Vol. 131, No. 4, July, 1984.

[44] H. Wunderlich and S. Hellebrand, "Tools and devices Supporting the Pseudo-Exhaustive Test," Proc. European Design Automation Conf., pp. 13-17, 1990.

[45] I. Shperling and E. J. McCluskey, "Circuit Segmentation for Pseudo-Exhaustive Testing via Simulated Annealing," Proc. Intl. Test Conf., pp. 58-66, 1987.

[46] J. G. Udell, "Test Set for Pseudo-exhaustive BIST," Proc. Int. Conf. Computer-Aided Design, pp. 52-55, 1986.

[47] D. Kagaris, F. Makedon, and S. Tragoudas, "On Minimization Hardware Overhead for Pseudo-Exhaustive Circuit Testability," Proc. IEEE Int'l Conf. Computer Design, pp. 358-364, 1992.

[48] J. G. Udell, " Efficient Segmentation for Pseudo-Exhaustive BIST," Proc. IEEE Custom Integrated Circuits, PP. 13.6.1-13.6.6, 1992.

[49] E. R. Barnes, "An Algorithm for Partitioning the Nodes of a Graph," SIAM J. Algebraic and Discrete Methods, Vol. 3, pp. 541-550, 1982.

[50] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell Systems Technical Journal, Vol. 49, pp. 291-307, 1970.

[51] W. E. Donath and A. J. Hoffman, "Lower Bounds for the Partitioning of Graphs," IBM J. Res. Development, PP. 420-425. 1973

[52] P. Bardell, W. McAnney, J. Savir, "Built-In Test for VLSI: Pseudorandom Techniques," John Wiley & Sons 1987.

[53] J. Savir, W. H. McAnney, S. R. Vecchio, "Testing for coupled cells in Random Access Memories," IEEE Int. Tst Conf., pp. 439-451, 1989.

[54] R. Dekker, F. Beenker, L. Thissen, "Fault Modeling and Test algorithm for Static Random Access Memories," Proc. Int. test Conf., pp. 343-352, 1988.

[55] O. Kebichi, M. Nicolaidis, "A tool for Automatic Generation of BISTed and Transparent BISTed RAMs," IEEE Int. test Conf., pp. 570-575, 1992.